COSE474-2024F: Deep Learning HW1

2022320077 김수현

2.1 Data Manipulation

In [4]:
```
import torch
```

In [5]:
```
x = torch.arange(12, dtype=torch.float32) #can create vector
x # showes element of vector x
```

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

In [6]:
```
x.numel() # total number of elements
```

Out[6]:
```
12
```

In [7]:
```
x.shape # matrix 12*1
```

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

In [8]:
```
X = x.reshape(3,4) # reshape 3*4
X
```

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

In [9]:
```
torch.zeros((2,3,4)) # set all zero
```

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]:
```
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]:
```
torch.randn(3,4) # random number
```

Out[11]:
```
tensor([[ 1.5556,  0.0024, -0.6005,  1.4857],
        [ 0.3021,  0.7770, -1.4833,  1.5595],
        [-0.2918, -0.0612, -0.5833,  1.5389]])
```

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

- 2.1.2 Indexing and Slicing

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

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

In [14]:
```
X[1,2] = 17 # rewrite the element
X
```

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

In [15]:
```
X[:2, :] = 12 # rewrite the matrix elements in two row
X
```

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

- 2.1.3 Operations

In [17]:
```
torch.exp(x) # natural number exponent
```

Out[17]:
```
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 [18]:
```
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 # basic calculation
```

Out[18]:
```
(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 [19]:
```
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) # along rows dim =0,
columns dim =1
```

Out[19]:
```
(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 [20]:
```
X == Y
```

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

In [21]:
```
X.sum()
```

Out[21]:
```
tensor(66.)
```

- 2.1.4. Broadcasting

In [23]:
```
a = torch.arange(3).reshape((3,1))
b = torch.arange(2).reshape((1,2))
a, b
```

Out[23]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

In [24]:
```
a + b
```

Out[24]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

- 2.1.5 Saving Memory

In [26]:
```
before = id(Y) # pointing the memory address
Y = Y + X # new address pop up
id(Y) == before
```

Out[26]:
```
False
```

In [27]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z);', id(Z))
```

```
id(Z): 5348166624
id(Z); 5348166624
```

In [29]:
```
before = id(X)
X += Y
id(X) == before
```

Out[29]:
```
True
```

- 2.1.6. Conversion to Other Python Objects

In [31]:
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

Out[31]:
```
(numpy.ndarray, torch.Tensor)
```

In [32]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[32]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

2.1 Discussion & Exercise

> Why the id of Z is not changed?

- 2.1.8 - 1

In [34]:
```
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)

X < Y
```

Out[34]:
```
tensor([[ True, False,  True, False],
        [False, False, False, False],
        [False, False, False, False]])
```

In [35]:
```
X > Y
```

Out[35]:
```
tensor([[False, False, False, False],
        [ True,  True,  True,  True],
        [ True,  True,  True,  True]])
```

- 2.1.8 -2

In [37]:
```
X = torch.arange(12, dtype=torch.float32).reshape((2,3,2))
Y = torch.randn(2,3,2)

X, Y, X == Y, X > Y, X < Y
```

Out[37]:
```
(tensor([[[ 0.,  1.],
          [ 2.,  3.],
          [ 4.,  5.]],

         [[ 6.,  7.],
          [ 8.,  9.],
          [10., 11.]]]),
 tensor([[[ 0.5685, -0.1041],
          [-0.8051, -0.9940],
          [-0.0774, -1.1406]],

         [[-1.7770, -0.9309],
          [ 0.9783, -0.1379],
          [-0.9575, -0.7157]]]),
 tensor([[[False, False],
          [False, False],
          [False, False]],

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

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

         [[False, False],
          [False, False],
          [False, False]]]))
```

In [38]:
```
X + Y
```

Out[38]:
```
tensor([[[ 0.5685,  0.8959],
         [ 1.1949,  2.0060],
         [ 3.9226,  3.8594]],

        [[ 4.2230,  6.0691],
         [ 8.9783,  8.8621],
         [ 9.0425, 10.2843]]])
```

In [39]:
```
X * Y
```

Out[39]:
```
tensor([[[  0.0000,  -0.1041],
         [ -1.6101,  -2.9819],
         [ -0.3098,  -5.7032]],

        [[-10.6619,  -6.5165],
         [  7.8262,  -1.2412],
         [ -9.5754,  -7.8726]]])
```

2.2. Data Preprocessing

- 2.2.1 Reading the Dataset

In [41]:
```
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 [42]:
```
import pandas as pd

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

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

- 2.2.2 Data Preparation

In [44]:
```
inputs, targets = data.iloc[:, 0:2], data.iloc[:,2] # iloc: integer-location based indexing
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 [45]:
```
inputs = inputs.fillna(inputs.mean()) # mean() to fill out with mean of datas
print(inputs)
```

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

- 2.2.3 Conversion to Tensor Format

In [47]:
```
import torch

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

Out[47]:
```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

- the result of the tesor is shown

false as 0 true as 1

2.2 Discussion & Exercise

> What is the main method that is used for image data or audio data?

In [51]:
```
import pandas as pd

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data"
columns = ["Sex", "Length", "Diameter", "Height", "Whole weight", "Shucked weight", "Viscera weight", "Shell weight", "Rings"]

# Read data .csv
abalone_data = pd.read_csv(url, names=columns)

missing_fraction = abalone_data.isnull().mean()

numerical_columns = abalone_data.select_dtypes(include=['float64', 'int64']).columns
categorical_columns = abalone_data.select_dtypes(include=['object']).columns

num_fraction = len(numerical_columns) / abalone_data.shape[1]
cat_fraction = len(categorical_columns) / abalone_data.shape[1]

print("Missing Value Fraction per Column:")
print(missing_fraction)
print(f"Fraction of Numerical Variables: {num_fraction}")
print(f"Fraction of Categorical Variables: {cat_fraction}")
```

```
Missing Value Fraction per Column:
Sex               0.0
Length            0.0
Diameter          0.0
Height            0.0
Whole weight      0.0
Shucked weight    0.0
Viscera weight    0.0
Shell weight      0.0
Rings             0.0
dtype: float64
Fraction of Numerical Variables: 0.8888888888888888
Fraction of Categorical Variables: 0.1111111111111111
```

In [52]:
```
#2

numerical_data = abalone_data[["Length", "Diameter", "Height"]]
categorical_data = abalone_data[["Sex"]]
```

```
print(numerical_data.head())
print(categorical_data.head())
```

```
   Length  Diameter  Height
0   0.455     0.365   0.095
1   0.350     0.265   0.090
2   0.530     0.420   0.135
3   0.440     0.365   0.125
4   0.330     0.255   0.080
  Sex
0   M
1   M
2   F
3   M
4   I
```

In [53]:
```
#3

import numpy as np

rows, cols = 1000000, 10
large_data = pd.DataFrame(np.random.randn(rows, cols), columns=
[f'col{i}' for i in range(cols)])

memory_usage = large_data.memory_usage(deep=True).sum() / (1024**2)
print(f"Memory usage for large dataset: {memory_usage:.2f} MB")
```

```
Memory usage for large dataset: 76.29 MB
```

When a dataset has a large number of categories, it can be challenging to encode them. One-hot encoding is common but can lead to high dimensionality. If the categories are too many, or if they are unique (like user IDs or product SKUs), you can:

- Label encode: Assign a numerical label to each category.
- Frequency encoding: Assign numerical labels based on frequency of each category.
- Dimensionality reduction: Use techniques like PCA on one-hot encoded vectors.

In [55]:
```
#4
from sklearn.preprocessing import LabelEncoder

# Example of Label Encoding
label_encoder = LabelEncoder()
abalone_data['Sex_encoded'] =
label_encoder.fit_transform(abalone_data['Sex'])
print(abalone_data[['Sex', 'Sex_encoded']].head())
```

```
  Sex  Sex_encoded
0   M            2
1   M            2
2   F            0
3   M            2
4   I            1
```

In [56]:
```
#5
import numpy as np
```

```
tensor = np.random.rand(101, 101)
np.save('tensor.npy', tensor)

loaded_tensor = np.load('tensor.npy')
print(loaded_tensor.shape)
```

```
(101, 101)
```

2.3 Linear Algebra

- 2.3.1 Scalars

In [58]:
```
import torch

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

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

Out[58]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

- 2.3.2 Vectors

In [60]:
```
x = torch.arange(3)
x # vector
```

Out[60]:
```
tensor([0, 1, 2])
```

In [61]:
```
x[2]
```

Out[61]:
```
tensor(2)
```

In [62]:
```
len(x) # check the dimensionality
```

Out[62]:
```
3
```

In [63]:
```
x.shape
```

Out[63]:
```
torch.Size([3])
```

- 2.3.3 Matrices

In [65]:
```
A = torch.arange(6).reshape(3,2)
A
```

Out[65]:
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

In [66]:
```
A.T
```

Out[66]:
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

In [67]:
```python
A = torch.tensor([[1,2,3], [2, 0, 4], [3, 4, 5]]) # symmetric matrix
print(A)
print(A.T)
A == A.T
```

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

Out[67]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

- 2.3.4 Tensors

In [69]:
```python
torch.arange(24).reshape(2,3,4)
```

Out[69]:
```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

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

- 2.3.5 Basic Properties

In [71]:
```python
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()
A, A + B
```

Out[71]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

In [72]:
```python
A * B
```

Out[72]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

In [73]:
```python
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Out[73]:
```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

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

- 2.3.6 Reduction

In [75]:
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

Out[75]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

In [76]:
```
A.shape, A.sum()
```

Out[76]:
```
(torch.Size([2, 3]), tensor(15.))
```

In [134…
```
A.shape, A.sum(axis=0).shape # to reduce in rows(axis 0) : output == column size
```

Out[134…
```
(torch.Size([2, 3]), torch.Size([3]))
```

In [136…
```
A.shape, A.sum(axis=1).shape # to reduce in columns(axis 1) : output == row size
```

Out[136…
```
(torch.Size([2, 3]), torch.Size([2]))
```

In [138…
```
A.sum(axis=[0, 1]) == A.sum()
```

Out[138…
```
tensor(True)
```

In [140…
```
A.mean(), A.sum() / A.numel()
```

Out[140…
```
(tensor(2.5000), tensor(2.5000))
```

In [142…
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

Out[142…
```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

- 2.3.7 Non-reduction Sum

In [147…
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

Out[147…
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

In [149…
```
A / sum_A
```

Out[149…
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

In [151…
```
A.cumsum(axis=0)
```

Out[151…
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

- 2.3.8 Dot Products

In [155…
```
y = torch.ones(3, dtype=torch.float32)
x, y, torch.dot(x, y)
```
Out[155…
```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```
In [157…
```
torch.sum(x * y)
```
Out[157…
```
tensor(3.)
```

- 2.3.9 Matrix-Vector Products

In [160…
```
A.shape, x.shape, torch.mv(A, x), A@x
```
Out[160…
```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 1
4.]))
```

- 2.3.10 Matrix-Matrix Multiplication

In [163…
```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```
Out[163…
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

- 2.3.11 Norms

In [166…
```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```
Out[166…
```
tensor(5.)
```
In [168…
```
torch.abs(u).sum()
```
Out[168…
```
tensor(7.)
```
In [170…
```
torch.norm(torch.ones((4,9)))
```
Out[170…
```
tensor(6.)
```

2.4 Discussion

> **How does each norms are used?**

as the texts said the norm of vector tells us the size of vector. Each norms has same purpose though, each one could be used in other aim

> **What's the difference between tensors and matrices**

Tensor : it can be used in any dimesional space(scala, vector, matrix, so on...) matrix : two 2-dimensional space

2.5 Automatic Differentiation

- 2.5.1 A simple Function

In [199…
```
import torch

x = torch.arange(4.0, requires_grad=True)
x.grad # gradient default is None
x
```

Out[199…
```
tensor([0., 1., 2., 3.], requires_grad=True)
```

In [201…
```
y = 2 * torch.dot(x,x)
y
```

Out[201…
```
tensor(28., grad_fn=<MulBackward0>)
```

In [203…
```
y.backward()
x.grad
```

Out[203…
```
tensor([ 0.,  4.,  8., 12.])
```

In [205…
```
x.grad == 4 * x
```

Out[205…
```
tensor([True, True, True, True])
```

In [209…
```
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

Out[209…
```
tensor([1., 1., 1., 1.])
```

- 2.5.2 Backward for Non-Scaler Variables

In [216…
```
x.grad.zero_()
y = x * x
y.sum().backward()
x.grad
```

Out[216…
```
tensor([0., 2., 4., 6.])
```

- 2.5.3 Detaching Computation

In [223…
```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x
```

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

Out[223…
```
tensor([True, True, True, True])
```

In [230…
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

Out[230…
```
tensor([True, True, True, True])
```

- 2.5.4 Gradients and Python Control Flow

In [239…
```
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 [241…
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In [243…
```
a.grad == d / a
```

Out[243…
```
tensor(True)
```

2.5 Discussion

- we need to attach gradient(make it True) to variables that we want to see
- then use backward method, access to result gradient
- with y.backward(), x.grad get the gradient of each elements
- backward method calculates the gradient

> **What is difference between detach and backward method?**

3.1 Linear Regression

In [5]:
```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

- 3.1.2 Vectirization for Speed

In [6]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [7]:
```
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[7]:
```
'0.06670 sec'
```

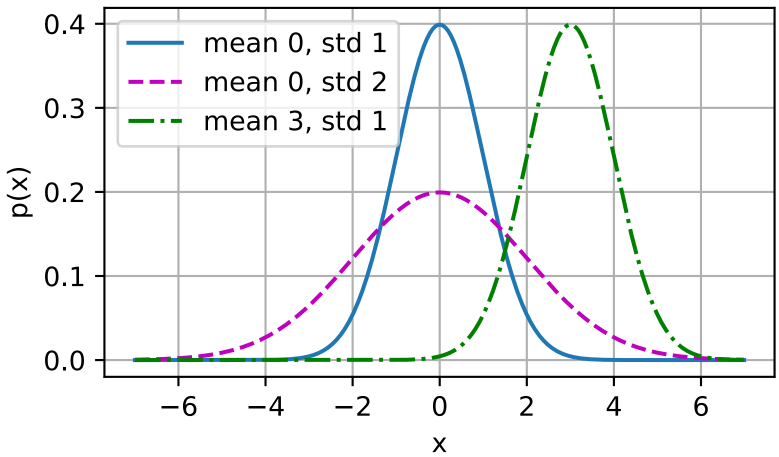
In [8]:
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[8]:
```
'0.00046 sec'
```

- 3.1.3 The Normal Distribution and Squared Loss

In [9]:
```
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 [10]:
```
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])
```



3.2 Object-oriented Design for Implementation

In [3]:
```
import time
import numpy as np
import torch
```

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

- 3.2.1. Utilities

In [4]:
```
def add_to_class(Class): #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

In [5]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In [6]:
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

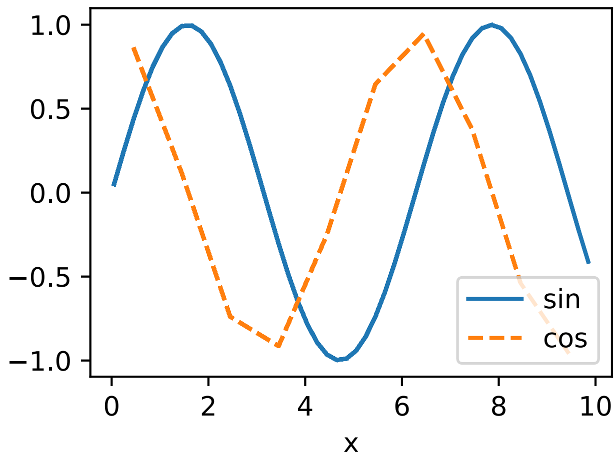
```
Class attribute "b" is 1
```

In [7]:
```
class HyperParameters:
    """The base class for hyperparameters."""
    def save_hyperprameters(self, ignore=[]):
        raise NotImplemented
```

In [8]:
```
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 [9]:
```
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

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



- 3.2.2 Models

In [10]:
```
class Module(nn.Module, d2l.HyperParameters): #@save
    """The base class of models."""
    def __init__(self, plot_treain_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
```

- 3.2.3 Data

In [11]:
```
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_datalpader(train=False)
```

- 3.2.4 Training

In [14]:
```
class Trainer(d2l.HyperParameters):
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

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

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

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

    def fit_epoch(self):
        raise NotImplementedError
```

3.2 Memo

- Module class is basic class for the all kinds of model
- **init** gets hyperparameters,
- for training step, accepts a data batch -> to return loss value
- for configure optimizers return optimize method
- (option) validation step : evaluation measure

3.4 Linear Regression Implementation from Scratch

In [14]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

- 3.4.1 Defining the model

In [15]:
```
class LinearRegressionScratch(d2l.Module): #@save
    """The linear regression model implmented for 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 [16]:
```
@d2l.add_to_class(LinearRegressionScratch) #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b #linear equation
```

- 3.4.2 Defining the Loss function

In [17]:
```
@d2l.add_to_class(LinearRegressionScratch) #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

- 3.4.3 Defining the Optimization Algorithm

In [21]:
```
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_()

@d2l.add_to_class(LinearRegressionScratch) #@save
```

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

- 3.4.4 Training
- initialize parameters

$$(w, b)$$

- compute gradient

$$\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \mathbf{w}, b)$$

- updata prarmerters

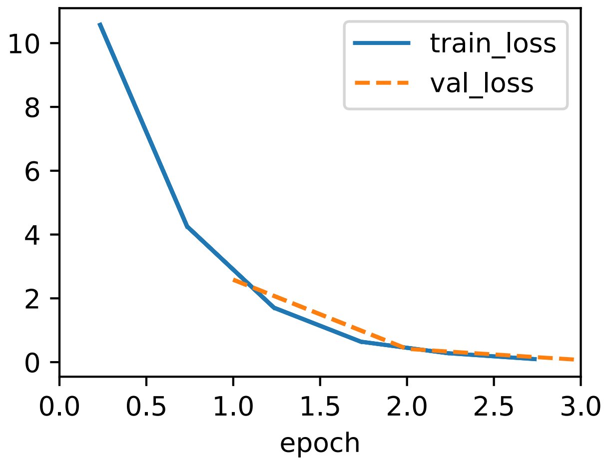
$$(\mathbf{w}, \mathbf{b}) \leftarrow (\mathbf{w}, \mathbf{b}) - \eta \mathbf{g}$$

In [22]:

```
@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 :
                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 [23]:

```
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 [25]:
```
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([ 0.1199, -0.2582])
error in estimating b: tensor([0.2438])
```

3.4 Memo

> Linear

- deviation **0.01** is a magic number
- **y_hat** is predicted value of true value y
- loss function return average loss value among all examples(instances) in the minibatch

> SDG

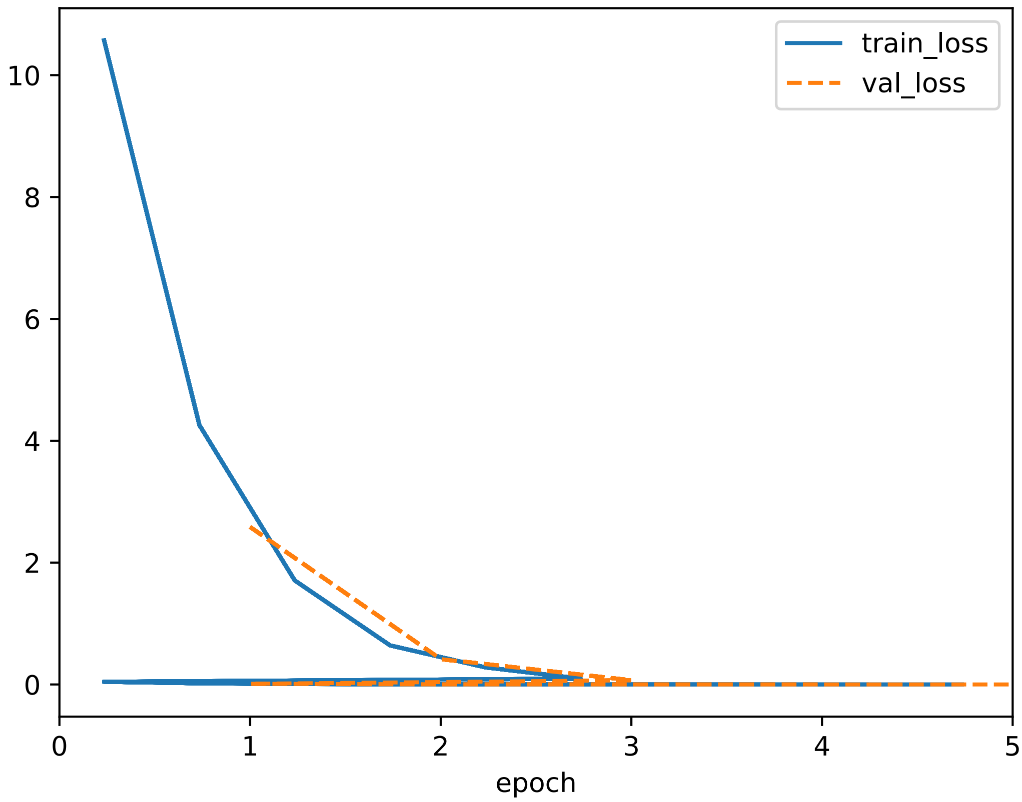
- Stochastic Gradient Descent
- get the random sample in training data
- calculate the loss function gradient
- learning rate(lr) is a important hyper parameter

> Epoch

- the number of the train with all the training set

3.4 Exercise

In [26]:
```
model_ex = LinearRegressionScratch(2, lr= 0.04)
data_ex = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=2.3)
trainer = d2l.Trainer(max_epochs=5)
trainer.fit(model, data)
```



4.1 Softmax Regression

4.1 Memo

- Regression is usually used for to answer **how much?** and **how many?**

> Soft Max

We have two problems directly using the output of the regression

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

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

$$\operatorname*{arg\,max}_j \hat{y}_j = \operatorname*{arg\,max}_j o_j.$$

> Vectorization

$$\mathbf{X} \in \mathbb{R}^{n\times d} \mathbf{W} \in \mathbb{R}^{d\times q} \mathbf{b} \in \mathbb{R}^{1\times q}.$$
$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b},$$
$$\hat{\mathbf{Y}} = \mathrm{softmax}(\mathbf{O}).$$

> Loss Function

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

$$= \sum_{j=1}^q y_j \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j$$

$$= \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j.$$

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

> Entropy : uncertainty of probability distribution

to quantify the amount of information contained in data is the central idea

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

4.1 Discussion

> **How can we interprete the entropy value**

if entropy is 0 : model can perfectly predict max entropy: not a good model

4.2 The Image Classification Dataset

In [27]:
```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

- 4.2.1 Loading the Dataset

In [29]:
```
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 [30]:
```
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/FashionMNIST/raw/train-images-idx3-ubyte.gz
100.0%
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/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100.0%
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/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100.0%
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/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100.0%
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
```

Out[30]:
```
(60000, 10000)
```

In [31]:
```
data.train[0][0].shape
```

Out[31]:
```
torch.Size([1, 32, 32])
```

In [33]:
```
@d2l.add_to_class(FashionMNIST) #@save
def text_labels(self, indices):
    """return text lables."""
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

- 4.2.2 Reading a Minibatch

In [35]:
```
@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 [36]:
```
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 [44]:
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

Out[44]: '4.38 sec'

- 4.2.4. Visualization

In [39]:
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
#@save
    """Plot a list of images."""
    raise NotImplementedError
```

In [43]:
```
@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)
```

4.2 Discussion

1. Does reducing the batch_size (for instance, to 1) affect the reading performance?
    - get slower training on GPU, faster iteration time, frequent parameter update etc.
2. What would be the best batch_size

- it doesn't have a typical answer
- generally: 32, 64 - for image : 128 or 256 , -for time series : 16 or 32

4.3 The base Classification Model

In [45]:
```
import torch
from d2l import torch as d2l
```

4.3.1 The Classifier Class

In [48]:
```
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 [49]:
```
@d2l.add_to_class(d2l.Module) #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

4.3.2 Accuracy

In [50]:
```
@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
```

4.4 Softmax Regression Implementation from Scratch

In [52]:
```
import torch
from d2l import torch as d2l
```

4.4.1 The Softmax

In [53]:
```
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[53]:
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

In [54]:
```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition
```

In [55]:
```
X = torch.rand((2,5))
```

In [56]:
```
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

Out[56]:
```
(tensor([[0.2240, 0.2018, 0.2228, 0.1939, 0.1576],
         [0.1652, 0.2281, 0.2474, 0.2058, 0.1534]]),
 tensor([1., 1.]))
```

- 4.4.2 The model

In [58]:
```
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 [59]:
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

- 4.4.3 The Cross-Entropy Loss

In [60]:
```
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[60]:
```
tensor([0.1000, 0.5000])
```

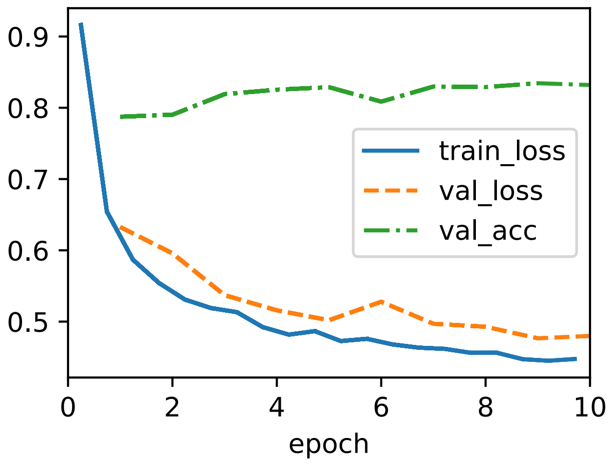
In [61]:
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[61]:
```
tensor(1.4979)
```

In [62]:
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

- 4.4.4 Training

In [63]:
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

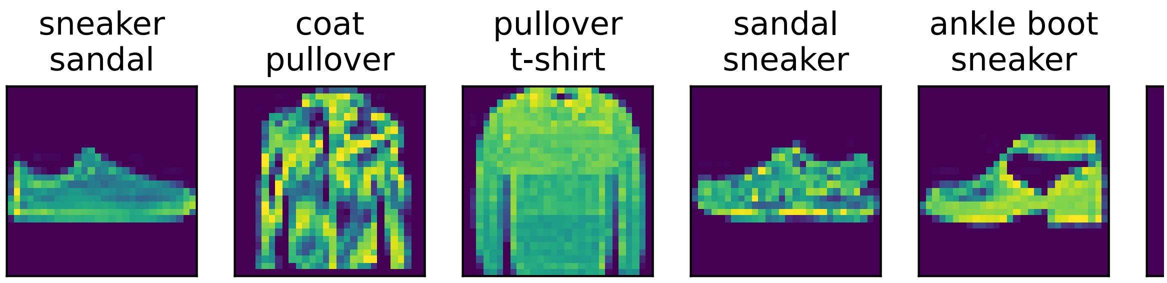


- 4.4.5 Prediction

In [64]:
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[64]:
```
torch.Size([256])
```

In [68]:
```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



4.4 Memo

> Requires of Computing the Softmax

- exponentiation of each term
- a sum over each row (to normalization constant for each instances)
- division of each row by its normalization constant(ensure that the result sums to 1)

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

5.1 Multilayer Perceptrons

In [69]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

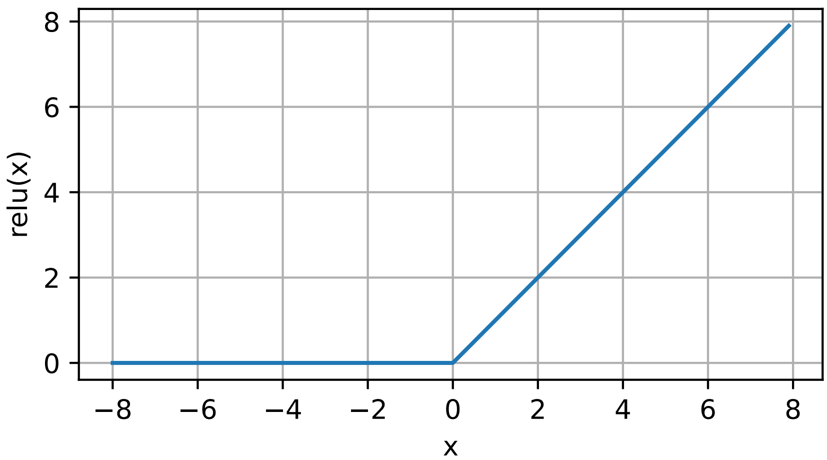
5.1 Memo

- linearity implies the weaker assumption of monotonicity (i.e., that any increase in our feature must either always cause an increase in our model's output, always cause a decrease in our model's output)
- we should consider which a linear model would be suitable
- With DNN we used observational data to jointly learn both a representation via hidden layers and a linear predictor that acts upon that representation.

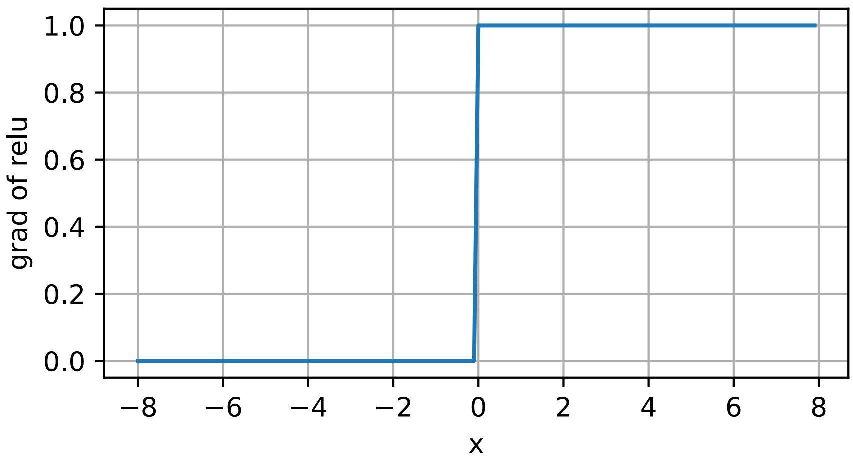
- 5.1.2 Activation Function

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

In [70]:
```
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))
```

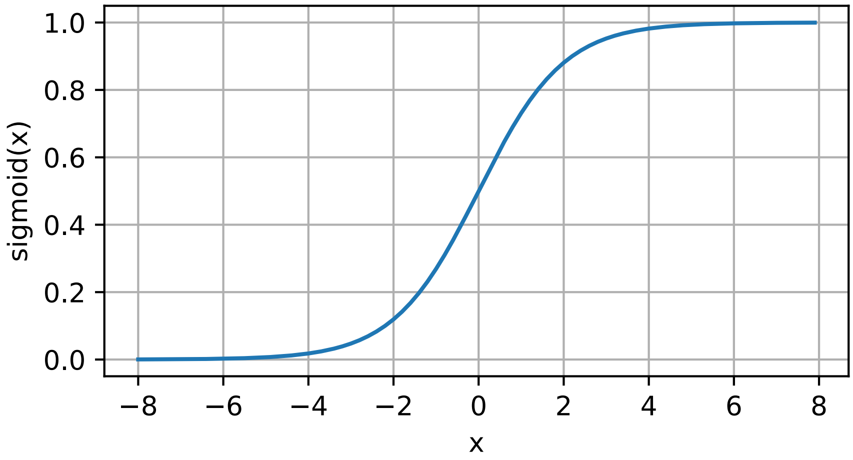


In [71]:
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5,2.5))
```



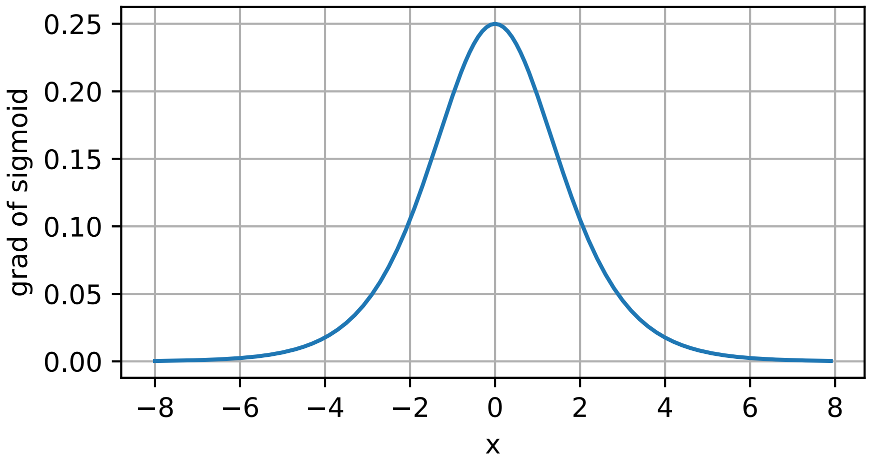
$$\mathrm{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

In [73]:
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



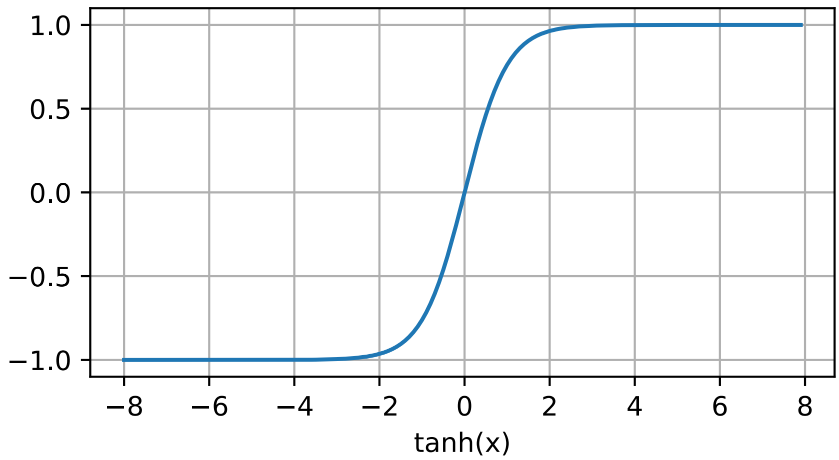
$$\frac{d}{dx} \mathrm{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \mathrm{sigmoid}(x)\left(1 - \mathrm{sigmoid}(x)\right).$$

In [74]:
```
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))
```



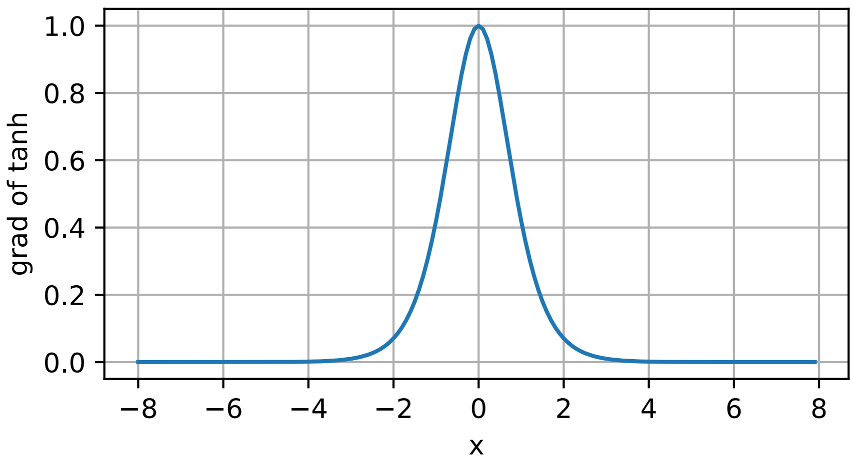
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

In [75]:
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'tanh(x)', figsize=(5,2.5))
```



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

In [77]:
```
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5,2.5))
```



5.1 Discussion

> How many hidden layers will be the best?(optimal number of hidden layers)

- one : simple problem
- two to four hidden layers : complex classification
- five or more : very complex tasks, especially when dealing with unstructured data like images, video, audio, or text (e.g., using CNNs or transformers)

> consideration

- many hidden layers can cause overfitting

5.2 Implemenatation of Multilayer Perceptrons

In [78]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

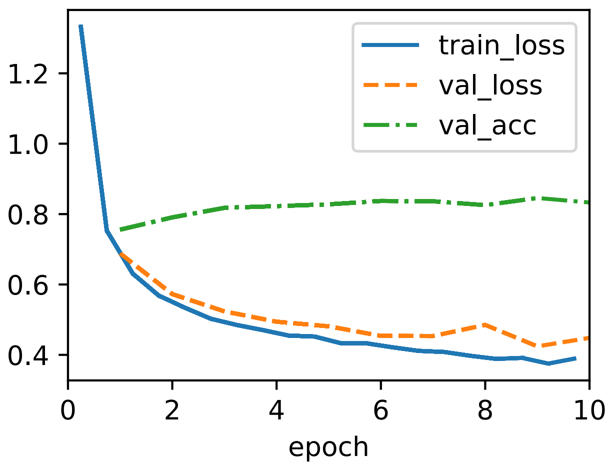
- 5.2.1 Implementation from Scratch

In [86]:
```
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 [87]:
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

In [88]:
```
@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 [89]:
```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256,
lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

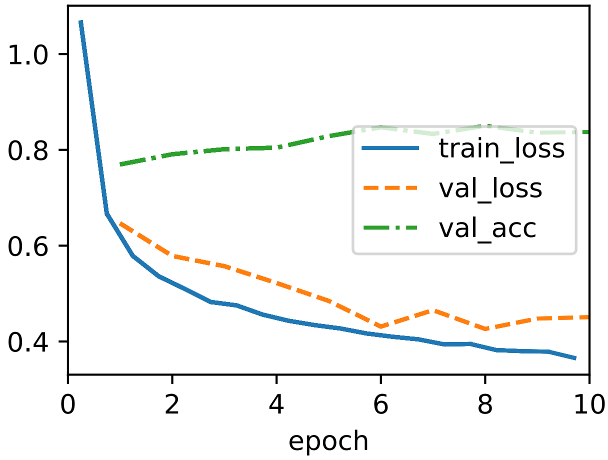


- 5.2.2 Concise Implementation

In [90]:
```
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 [91]:
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



5.3 Forward Propagation, Backward Propagation, and Computational Graphs

5.3 Memo

- When training NN, forward and backward propagation depend on each other.
- backpropagation reuses intermediate values from forward propagation to avoid duplicate calculations, so it should be stored.
- the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to out-of-memory errors.

5.3 Discussion & Exercise

- What are the advantages and disadvantages over training on a smaller minibatch?(related to 4.2 discussion)
  - advantage: lower memory use, faster initial convergence, better generalization due to noisy gradients, ideal for real-time
  - disadvantage: can cause instability, and slower overall convergence, inefficient for GPU, batch normalization would be less effective, require additional tuning
- As the purpose of Backward Propagation is decreasing loss, how can be possible?
-
  - Gradient Descent is used for decreasing loss. As gradient means the rate of change in loss function, backpropagation use gradient to modify the weight.
  - large gradient : loss is rapidly decreasing, revise weight more significantly.
  - small gradient : weight updates smaller and smoothly