Deep Learning HW 1 Summary

Chapter 2 Preliminaries

2.1 Data Manipulation

In [168…
```
import torch
```

In [169…
```
x = torch.arange(12, dtype=torch.float32)
x
```

Out[169…
```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

In [170…
```
x.numel()
```

Out[170…
```
12
```

In [171…
```
x.shape
```

Out[171…
```
torch.Size([12])
```

In [172…
```
X = x.reshape(3, 4)
X
```

Out[172…
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

In [173…
```
torch.zeros((2, 3, 4))
```

Out[173…
```
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 [174…
```
torch.ones((2, 3, 4))
```

Out[174…
```
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 [175…
```
torch.randn(3, 4)
```

Out[175…
```
tensor([[-0.6250,  0.2757, -1.5338,  1.6903],
        [ 0.9561,  2.0834,  0.0575, -0.2348],
        [ 1.2329,  1.0762, -1.3042, -2.1466]])
```

In [176…

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

Out[176…

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

In [177…

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

Out[177…

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

In [178…

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

Out[178…

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

In [179…

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

Out[179…

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

In [180…

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

Out[180…

```
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 [181…

```
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[181…

```
(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 [182…

```
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[182…

```
(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 [183…

```
X == Y
```

Out[183…
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

In [184…
```
X.sum()
```

Out[184…
```
tensor(66.)
```

In [185…
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

Out[185…
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

In [186…
```
a + b
```

Out[186…
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

In [187…
```
before = id(Y)
Y = Y + X
id(Y) == before
```

Out[187…
```
False
```

In [188…
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 2070672315472
id(Z): 2070672315472
```

In [189…
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 2070672316272
id(Z): 2070672316272
```

In [190…
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

Out[190…
```
(numpy.ndarray, torch.Tensor)
```

In [191…
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[191…
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

Exercise part in the d2l book of 2.1

In [192…

```
#ex1
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]])
print(X < Y, X > Y)
```

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

In [193…

```
#ex2
print(X + Y, X - Y, X * Y, X / Y) ##elementwise operation
```

```
tensor([[ 2.,  2.,  6.,  6.],
        [ 5.,  7.,  9., 11.],
        [12., 12., 12., 12.]]) tensor([[-2.,  0., -2.,  0.],
        [ 3.,  3.,  3.,  3.],
        [ 4.,  6.,  8., 10.]]) tensor([[ 0.,  1.,  8.,  9.],
        [ 4., 10., 18., 28.],
        [32., 27., 20., 11.]]) tensor([[ 0.0000,  1.0000,  0.5000,  1.0000],
        [ 4.0000,  2.5000,  2.0000,  1.7500],
        [ 2.0000,  3.0000,  5.0000, 11.0000]])
```

2.2 Data Preprocessing

In [194…

```
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 [195…

```
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 [196…

```
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 [197…

```
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 [198…

```
import torch

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

Out[198…

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

Exercise part in the d2l book of 2.2

In [199…

```
#ex1

#pip install ucimlrepo

from ucimlrepo import fetch_ucirepo

# fetch dataset
abalone = fetch_ucirepo(id=1)

# data (as pandas dataframes)
X = abalone.data.features
y = abalone.data.targets

df = pd.concat((X, y), axis=1)
print(df.head())
print(df.isnull().values.any()) ## no missing values

# variable information
print(abalone.variables) ## one feature is categorical, the others are numerical
```

```
   Sex  Length  Diameter  Height  Whole_weight  Shucked_weight  Viscera_weight  \
0    M   0.455     0.365   0.095        0.5140          0.2245          0.1010
1    M   0.350     0.265   0.090        0.2255          0.0995          0.0485
2    F   0.530     0.420   0.135        0.6770          0.2565          0.1415
3    M   0.440     0.365   0.125        0.5160          0.2155          0.1140
4    I   0.330     0.255   0.080        0.2050          0.0895          0.0395

   Shell_weight  Rings
0         0.150     15
1         0.070      7
2         0.210      9
3         0.155     10
4         0.055      7
False
             name     role         type demographic  \
0             Sex  Feature  Categorical        None
1          Length  Feature   Continuous        None
2        Diameter  Feature   Continuous        None
3          Height  Feature   Continuous        None
4    Whole_weight  Feature   Continuous        None
5  Shucked_weight  Feature   Continuous        None
6  Viscera_weight  Feature   Continuous        None
7    Shell_weight  Feature   Continuous        None
8           Rings   Target      Integer        None

                description  units missing_values
0         M, F, and I (infant)   None             no
1     Longest shell measurement     mm             no
2       perpendicular to length     mm             no
3            with meat in shell     mm             no
4                 whole abalone  grams             no
5                weight of meat  grams             no
6  gut weight (after bleeding)  grams             no
7             after being dried  grams             no
8  +1.5 gives the age in years   None             no
```

In [200…

```
#ex2
print(data)
inputs, targets = data.loc[:, ("NumRooms", "RoofType")], data.loc[:, "Price"] #N
inputs = pd.get_dummies(inputs, dummy_na=True) # Same result
print(inputs)
```

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

3. Possibly around millions of lines. The main limitation would possibly be memory, since many of pandas functions copy memory space rather than in-place replacement.

4. If there are too many categories, categories with similliar features could be regrouped.

5. The alternative to pandas could be numpy as mentioned in book, or even excel in case that is not Python based.

2.3 Linear Algebra

In [201…
```
import torch
```

In [202…
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

Out[202…
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

In [203…
```
x = torch.arange(3)
x
```

Out[203…
```
tensor([0, 1, 2])
```

In [204…
```
x[2]
```

Out[204…
```
tensor(2)
```

In [205…
```
len(x)
```

Out[205…
```
3
```

In [206…
```
x.shape
```

Out[206…
```
torch.Size([3])
```

In [207…
```
A = torch.arange(6).reshape(3, 2)
A
```

Out[207…
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

In [208…
```
A.T
```

Out[208…
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

In [209…
```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

Out[209…
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

In [210…
```
torch.arange(24).reshape(2, 3, 4)
```

Out[210…
```
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 [211…
```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()
A, A + B
```

Out[211…
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

In [212…
```
A * B
```

Out[212…
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

In [213…
```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Out[213…
```
(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 [214…
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

Out[214…
```
(tensor([0., 1., 2.]), tensor(3.))
```

In [215…
```
A.shape, A.sum()
```

Out[215…
```
(torch.Size([2, 3]), tensor(15.))
```

In [216…
```
A.shape, A.sum(axis=0).shape
```

Out[216…
```
(torch.Size([2, 3]), torch.Size([3]))
```

In [217…
```
A.shape, A.sum(axis=1).shape
```

Out[217…
```
(torch.Size([2, 3]), torch.Size([2]))
```

In [218…
```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

Out[218…
```
tensor(True)
```

In [219…
```
A.mean(), A.sum() / A.numel()
```

Out[219… (tensor(2.5000), tensor(2.5000))

In [220…
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

Out[220… (tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))

In [221…
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

Out[221…
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

In [222…
```
A / sum_A
```

Out[222…
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

In [223…
```
A.cumsum(axis=0)
```

Out[223…
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

In [224…
```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

Out[224… (tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))

In [225…
```
torch.sum(x * y)
```

Out[225… tensor(3.)

In [226…
```
A.shape, x.shape, torch.mv(A, x), A@x
```

Out[226… (torch.Size([2, 3]), torch.Size([3]), tensor([5., 14.]), tensor([5., 14.]))

In [227…
```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

Out[227…
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

In [228…
```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

Out[228… tensor(5.)

In [229…
```
torch.abs(u).sum()
```

Out[229… tensor(7.)

In [230…
```
torch.norm(torch.ones((4, 9)))
```

Out[230… tensor(6.)

Exercise part in the d2l book of 2.3

In [231…
```python
A = torch.arange(24).reshape(2, 3, 4)
B = torch.arange(1, 25).reshape(2, 3, 4)
C = torch.arange(9).reshape(3, 3)
print(A == (A.T).T, A.T + B.T == (A + B).T, (C + C.T) == (C + C.T).T)
print(len(A)) # first dimension
print(C / C.sum(axis = 1)) #normalization per axis 1
```

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

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

        [[True, True],
         [True, True],
         [True, True]],

        [[True, True],
         [True, True],
         [True, True]],

        [[True, True],
         [True, True],
         [True, True]]]) tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
2
tensor([[0.0000, 0.0833, 0.0952],
        [1.0000, 0.3333, 0.2381],
        [2.0000, 0.5833, 0.3810]])
```

7. Manhattan distance is l1 distance

Computation and memory usage can be differ which matrix to calculate first, due to the properties of matrix multiplication

In [232…
```python
#dimension reduction along that axis
import numpy as np
print(A, np.linalg.norm(A)) #every dimension
```

```
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]]]) 65.75712889109438
```

2.5 Data Manipulation

In [233…
```python
import torch
```

In [234…
```python
x = torch.arange(4.0)
x
```

Out[234…
```
tensor([0., 1., 2., 3.])
```

In [235…
```python
x.requires_grad_(True)
x.grad
```

In [236…
```python
y = 2 * torch.dot(x, x)
y
```

Out[236…
```
tensor(28., grad_fn=<MulBackward0>)
```

In [237…
```python
y.backward()
x.grad
```

Out[237…
```
tensor([ 0.,  4.,  8., 12.])
```

In [238…
```python
x.grad == 4 * x
```

Out[238…
```
tensor([True, True, True, True])
```

In [239…
```python
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

Out[239…
```
tensor([1., 1., 1., 1.])
```

In [240…
```python
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```

Out[240…
```
tensor([0., 2., 4., 6.])
```

In [241…
```python
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

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

Out[241…
```
tensor([True, True, True, True])
```

In [242…
```python
x.grad.zero_()
y.sum().backward()
x.grad == 2*x
```

Out[242…
```
tensor([True, True, True, True])
```

In [243…
```python
def f(a):
    b = a * 2
```

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

In [244…

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

In [245…

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

Out[245…

```
tensor(True)
```

Exercise part in the d2l book of 2.5

1. Second derivation's size is square compare to that of first deriviate.

2. Running backpropagation twice throws errors because torch frees the forward calculation memory.

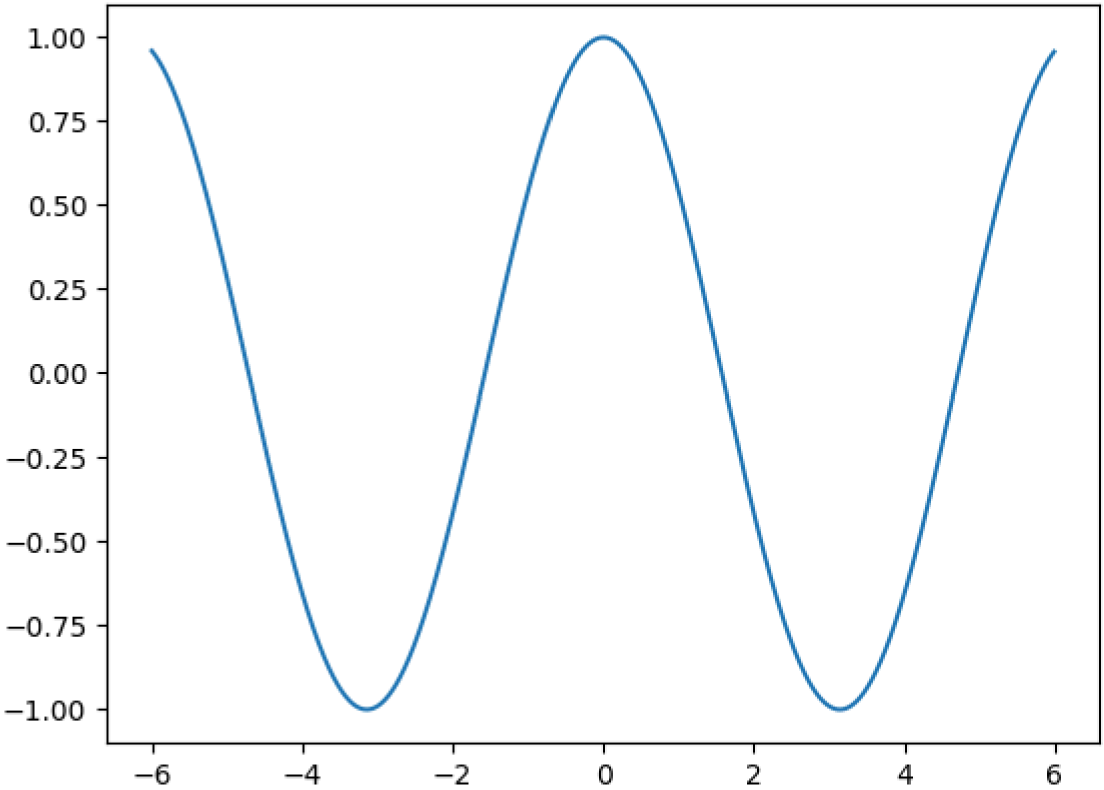
3. If a wasn't scalar, still it would produce same effect except that divide by a is replaced by inverse.

4. Backward differentiation is used in backpropagation because it reduces the computation time, since calculating from scalar is more efficient.

In [246…

```
#ex 4
import matplotlib.pyplot as plt
import torch

x = torch.arange(start=-6, end=6, step=1e-2)
x.requires_grad_(True)
y = torch.sin(x)
y.backward(gradient=torch.ones(len(x)))
plt.plot(x.detach().numpy(), x.grad)
print(x, x.grad)
```

```
tensor([-6.0000, -5.9900, -5.9800,  ...,  5.9700,  5.9800,  5.9900],
       requires_grad=True) tensor([0.9602, 0.9573, 0.9544,  ..., 0.9514, 0.9544,
0.9573])
```



Chapter 2 discussion

2.1 Saving memory: Python allocate memory when we do operations like Y = Y + X to create newly updated Y

2.2 Pandas is data handing tool or library. Get_dummies function make categorical feature into one-hot vector.

2.3 In Python, the indexing is a bit different from usual mathmatical notation. One case would be zero and one indexing difference, and the order of dimensions. Basically normal arithmetic operations are elementwise in Torch and numpy, and most operations can be broadcasted in Torch as in numpy. Only difference is the shape of matrix or vector or scalar and matching their shape when calculating.

2.5 Torch handles differentiations using forward and backward mode. Torch needs gradient vector, which possiblely reflected the deep learning backpropagation process. Torch supports partial backward gradient calculation by detach.

In [247…

```
print(x.numel(), len(x)) # torch supports some built-in functions
```

```
1200 1200
```

Chapter 3 Linear Neural Networks for Regression

3.1 Linear regression

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

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

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

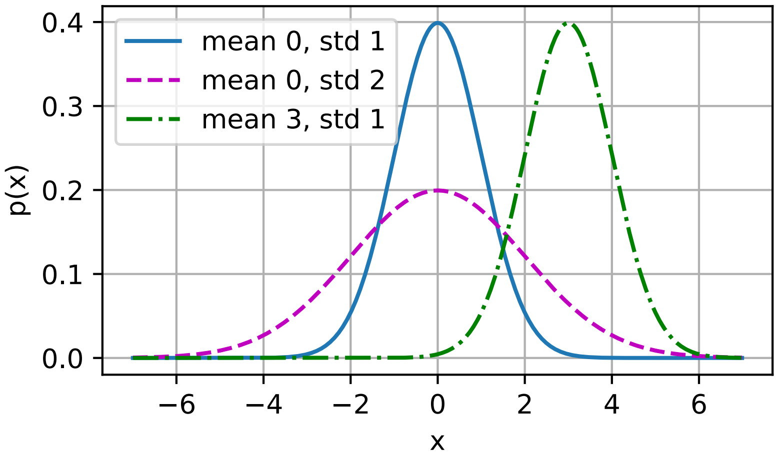
Out[3]: '0.21820 sec'

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

Out[4]: '0.00100 sec'

In [5]:
```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

In [6]:
```
# 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])
```



Some Exercises of 3.1 in the d2l book

1. If we deriviate the term, it follows that be should be the mean of data set {x_i}.

2. Bias term can be incoporated into weight matrix.

3. We can create square term and use linear regression as normal.

4. if X is not a full rank, we can remove linearly dependent training data. Or adding very small noise will make X not dependent.

5. MLP requires non linearlity to express every functions.

7, 8 Logarithmic price can represent the percentage change into addictive operaton with range of real number. Poisson distribution works for positive integer cases, contrary to gaussian distribution.

3.2 Object-Orientation Design for Implementation

In [7]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

In [8]:
```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

In [9]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In [10]:
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)
```

```
a.do()
```

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

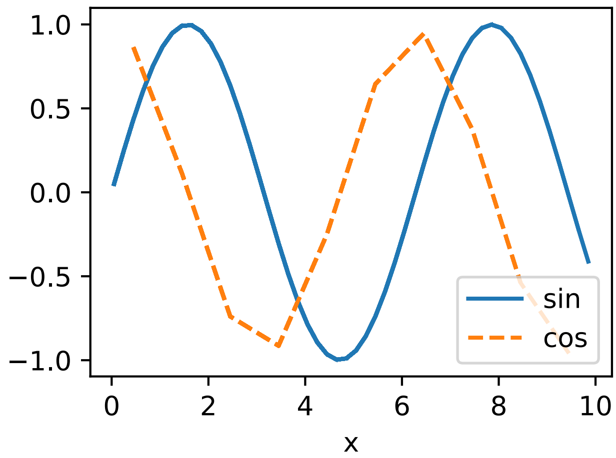
In [11]:
```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

In [12]:
```
# 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 [13]:
```
class ProgressBoard(d2l.HyperParameters):
    """The board that plots data points in animation."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2', 'C3'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

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

In [14]:
```
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 [15]:
```
class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

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

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

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

In [16]:
```
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
```

Some Exercises of 3.2 in the d2l book

In [17]:
```
#ex 2
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
```

3.4 Linear Regression Implementation from Scratch

In [18]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

In [19]:
```
class LinearRegressionScratch(d2l.Module):
    """The linear regression model implemented from scratch"""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

In [20]:
```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

In [21]:
```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

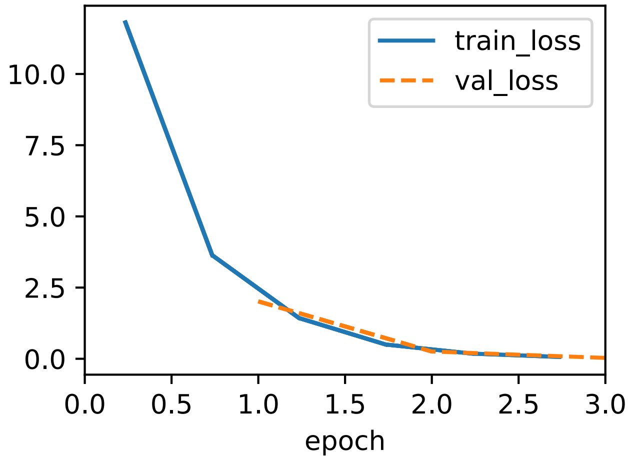
In [22]:
```
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

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

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

In [23]:
```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

In [24]:
```
@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    return batch

@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:
                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 [25]:
```
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 [26]:
```
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([ 0.0725, -0.1359])
error in estimating b: tensor([0.2002])
```

Some Exercises of 3.4 in the d2l book

1. if weight is set to zero, the gradient would be zero, which stops the learning process. other weights will be fine.

2, 3 Possibly No to Ohm's law if we consider very general (function) relations and Yes to applied in Planck's law.

4. Calculating second deriviate requires square operation (Hessian) and calculating inverse matrix is also a problem if we want to use it for Newton's method that I've heard of. There are some approximation algorithm that is known.

5. Too large learning rate causes oscillation, and too small learning rate causes slow training. Adding epoch usually reduces error.

6. last iteration will contain less training points than usual minibatch.

7. L2 loss is more sensitive to outliers, and L1 is more robust to that. We can combine both loss by possibly adding l1 regularization to simple linear regression model, though it is slightly deviates from the concept of absolute value loss.

8. We need to reshuffle the dataset because the model can memorize the training set.

Chapter 3 discussion

3.1 Linear regression is based on assumptions that expected value of target can be represented by weighted linear sum plus bias, and error is gaussian distribution. From that assumption, the square sum loss function is most natural scoring function. Vectorized code is more faster because torch or numpy can internally use faster c based code.

3.2 By inheriting and reusing the functions in a form of OOP design, solving machine learning or deep learning problems from already well built library is much easier with less modification required. Python setattr enables to set functions as methods after class is created or defined.

3.4 Minibatch SGD works in many other optimization problems, though it doesn’t guarantees best accuracy or minimum loss value.

Chapter 4 Linear Neural Networks for Classification

4.1 Softmax Regression

Softmax regression is a classification model that uses linear regression and transform its value to make a log likelihood, probablistic prediction for each class. It uses one hot vector encoding, and cross entropy loss which is negative sum of probablity times log probablity as the name entropy indicates.

4.2 The Image Classification Dataset

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

d2l.use_svg_display()
```

In [2]:
```
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

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

Out[3]: (60000, 10000)

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

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

In [5]:
```
@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 [6]:
```
@d2l.add_to_class(FashionMNIST)
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
```

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

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

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

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

In [9]:
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """Plot a list of images."""
    raise NotImplementedError
```

In [10]:
```
@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=[]):
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```

```
<Figure size 1200x150 with 8 Axes>
```

Some Exercises of 4.2 in the d2l book

1. Reducing batch_size to 1 would decrease the reading performance because image should be read one by one.

4.3. The Base Classification Model

In [11]:
```
import torch
from d2l import torch as d2l
```

In [12]:
```
class Classifier(d2l.Module):
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
```

In [13]:
```
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

In [14]:
```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the number of correct predictions."""
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

4.4. Softmax Regression Implementation from Scratch

In [15]:
```
import torch
from d2l import torch as d2l
```

In [16]:
```
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[16]:
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

In [17]:
```
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 [18]:
```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

Out[18]:
```
(tensor([[0.2386, 0.2324, 0.2046, 0.1845, 0.1400],
         [0.1764, 0.1137, 0.2710, 0.1537, 0.2852]]),
 tensor([1.0000, 1.0000]))
```

In [19]:
```
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 [20]:
```
@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 [21]:
```python
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[21]:
```
tensor([0.1000, 0.5000])
```

In [22]:
```python
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[22]:
```
tensor(1.4979)
```

In [23]:
```python
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

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

```
<Figure size 350x250 with 1 Axes>
```

In [25]:
```python
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[25]:
```
torch.Size([256])
```

In [26]:
```python
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)
```

```
<Figure size 1200x150 with 8 Axes>
```

Chapter 4 discussion

Softmax regressions works like a linear regression. The linear multiplilcation and bias added value is transformed using softmax function, which represent probablistic interpretation and enable classification by finding argmax class. FashionMNIST model is more general dataset version of MNIST, and we can check that simple linear model with softmax works well with validation accuracy around 80%. The caveat is, FashionMNIST problem still might be too easy problem for deep learning models.

Chapter 5 Preliminaries

5.1. Multilayer Perceptrons

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

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

```
<Figure size 500x250 with 1 Axes>
```

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

```
<Figure size 500x250 with 1 Axes>
```

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

```
<Figure size 500x250 with 1 Axes>
```

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

```
<Figure size 500x250 with 1 Axes>
```

In [6]:
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

```
<Figure size 500x250 with 1 Axes>
```

In [7]:
```
# 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))
```

```
<Figure size 500x250 with 1 Axes>
```

Some Exercises of 5.1 in the d2l book

1. Affine transformation multiple times still parts of Affine function. Zero multiplication trivially reduces the network to express single number.
2. Since Relu or pRelu split function into two part, linearity will still be conserved, with splitted or piecewise,
3. gradient vanishing is common for sigmoid function, one such example would be why we devised LSTM rather than RNN.

In [15]:
```
y = torch.prelu(x, torch.tensor(0.1))
d2l.plot(x.detach(), y.detach(), 'x', 'pRelu(x)', figsize=(5, 2.5))
```

```
<Figure size 500x250 with 1 Axes>
```

In [14]:
```
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))
```

```
<Figure size 500x250 with 1 Axes>
```

In [16]:
```
y = torch.mul(x, torch.sigmoid(2*x))
d2l.plot(x.detach(), y.detach(), 'x', 'Swish activation function', figsize=(
```

```
<Figure size 500x250 with 1 Axes>
```

In [17]:
```
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))
```

```
<Figure size 500x250 with 1 Axes>
```

In [ ]:

5.2 Implementation of Multilayer Perceptrons

In [30]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

In [31]:
```
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) * sigm
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

In [32]:
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

In [33]:
```
@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 [34]:
```
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)
```

```
<Figure size 350x250 with 1 Axes>
```

In [35]:
```
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 [65]:
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```

```
<Figure size 350x250 with 1 Axes>
```

Some Exercises of 5.2 in the d2l book

3. Because it reduces previous layer's information into single number.
4. Usually power of two is known to be best because of the cpu and gpu architecture, especially memory usage when models like transformer takes lots of memory to train.
5. Some weights initialization is known to work well in practice, like xavier initialization. Good initialization helps model to converge fast and possibly reach better accuracy.

In [141]:
```
```

```
tensor(0.8564)
```

```
<Figure size 350x250 with 1 Axes>
```

In [152]:
```
# ex 1
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))

accuracy = []
for num_hiddens in [64, 128, 256, 512]:
    model = MLP(num_outputs=10, num_hiddens=torch.tensor(num_hiddens, dtype=
    data = d2l.FashionMNIST(batch_size=256)
    trainer = d2l.Trainer(max_epochs=10)
    trainer.fit(model, data)
    data = d2l.FashionMNIST(batch_size=10000)
    X, y = next(iter(data.val_dataloader()))
    y_hat = model.net(X)
    acc = model.accuracy(y_hat, y)
    #print(f"{num_hiddens} hidden units acc:", acc)
    accuracy.append(acc)
```

```
<Figure size 350x250 with 1 Axes>

<Figure size 350x250 with 1 Axes>

<Figure size 350x250 with 1 Axes>

<Figure size 350x250 with 1 Axes>
```

In [155]:
```
max_idx = torch.argmax(torch.tensor(accuracy))
print(accuracy[max_idx], [64, 128, 256, 512][max_idx]) # generally more para
```

```
tensor(0.8535) 512
```

In [164]:

```
#ex 2
accuracy = []

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(64),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))


    model = MLP(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)

    data = d2l.FashionMNIST(batch_size=10000)
    X, y = next(iter(data.val_dataloader()))
    y_hat = model.net(X)
    acc = model.accuracy(y_hat, y)
```

```
<Figure size 350x250 with 1 Axes>
```

In [165]:

```
print(acc)
```

```
tensor(0.8535)
```

In [166]:
```python
# ex 3
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))

accuracy = []
for lr in [0.01, 0.03, 0.1, 0.3]:
    model = MLP(num_outputs=10, num_hiddens=256, lr=lr)
    data = d2l.FashionMNIST(batch_size=256)
    trainer = d2l.Trainer(max_epochs=10)
    trainer.fit(model, data)
    data = d2l.FashionMNIST(batch_size=10000)
    X, y = next(iter(data.val_dataloader()))
    y_hat = model.net(X)
    acc = model.accuracy(y_hat, y)
    #print(f"{num_hiddens} hidden units acc:", acc)
    accuracy.append(acc)

max_idx = torch.argmax(torch.tensor(accuracy))
print(accuracy[max_idx], [0.01, 0.03, 0.1, 0.3][max_idx])
```

```
tensor(0.8632) 0.3

<Figure size 350x250 with 1 Axes>

<Figure size 350x250 with 1 Axes>

<Figure size 350x250 with 1 Axes>

<Figure size 350x250 with 1 Axes>
```

5.3 Data Manipulation

Some Exercises of 5.3 in the d2l book

1. Gradient has same dimensionality of input.
2. Bias term just accepts upstream gradient (since local gradient is 1) and left no downstream gradient.
3. Forward computation takes about parameter memory and backpropagation takes about twice of it.

Chapter 5 discussion

5.1 GELU is x times cdf of normal distribution. As mentioned in the book, GELU is an alternative to ReLU. It has more smooth function, differentiable everywhere, has no dead ReLU effect. It is used in more recent deep learning architecture.

5.2 Adding more hidden layer usually increases model accuracy.

5.3 Backpropagation is the efficient method that enables to calculate gradient. It uses chain rule and computational graph to derive gradient of every parameters.

In [163]:
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.nn.GELU()
y = y(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```

```
<Figure size 500x250 with 1 Axes>
```

In [ ]:
```
```