COSE474 2024 Fall : Deep Learning HW1

0.1 Installation

```
1 pip install d2l==1.0.3
```

```
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0->d2l==1.0.3) (0.2.0
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0->d2l==1.0.3) (7.34.0
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0->d2l==1.0.3) (6.1.12
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.0->d2l==1.0.3) (6.3.3)
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-packages (from ipywidgets->jupyter==1.0.0->d2l==1.0
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ipywidgets->jupyter==1.0.0->d2l==1.0
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from jupyter-console->
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from jupyter-console->jupyter==1.0.0->d2l==1.0.3) (2.18.0
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (4.9.4)
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (4.12.3
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (6.1.0)
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (0.7.1)
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (0.4
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (3.1.4)
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (5.7
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (0
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (2.1.5
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (0.8
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (0.10.0
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (5.10.4)
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==1.0.3) (1.3.0)
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (24.0.1)
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (23.1.0)
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (1.6.0
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (1.8.3
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (0.18.
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (0.20
Requirement already satisfied: nbclassic>=0.4.7 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0->d2l==1.0.3) (1.1.0
Requirement already satisfied: qtpy>=2.4.0 in /usr/local/lib/python3.10/dist-packages (from qtconsole->jupyter==1.0.0->d2l==1.0.3) (2.4.1)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2
Requirement already satisfied: jedi>=0.16 in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l==1.0
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l==1.0
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l==1
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l==1.0.3
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dist-packages (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l==1
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.10/dist-packages (from jupyter-core>=4.7->nbconvert->jupyter==1.0.0
Requirement already satisfied: notebook-shim>=0.2.3 in /usr/local/lib/python3.10/dist-packages (from nbclassic>=0.4.7->notebook->jupyter==1.0
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/python3.10/dist-packages (from nbformat>=5.1->nbconvert->jupyter==1.0.0-
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.10/dist-packages (from nbformat>=5.1->nbconvert->jupyter==1.0.0->d2l=
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->jupyter-
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.10/dist-packages (from terminado>=0.8.3->notebook->jupyter==1.0.0->d2l==1
Requirement already satisfied: argon2-cffi-bindings in /usr/local/lib/python3.10/dist-packages (from argon2-cffi->notebook->jupyter==1.0.0->d2
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages (from beautifulsoup4->nbconvert->jupyter==1.0.0->d2l==
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages (from bleach->nbconvert->jupyter==1.0.0->d2l==1.0.3) (0
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ipython>=5.0.0->ipykernel->jup
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat>=5.1->nbconvert->jupy
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbforma
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat>=5.1->nbconvert-
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat>=5.1->nbconvert->jupy
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/python3.10/dist-packages (from notebook-shim>=0.2.3->nbclassic>=0.4.7-
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from argon2-cffi-bindings->argon2-cffi->notebook->jupy
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0.1->argon2-cffi-bindings->argon2-cffi->note
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>=1.8->notebook-shim>=0.2.3->
Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages (from jupyter-server<3,>=1.8->notebook-shim>=0.2.3-
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server<3,>=1.8->noteboo
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0->jupyter-server<3,>=1.8->noteb
```

2.1. Data Manipulation

2.1.1. basic

```
1 import torch
```

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

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

```
1 x.numel()
```

```
12
```

```
1 x.shape
```

```
torch.Size([12])
```

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

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

```
1 X.shape
```

```
torch.Size([3, 4])
```

```
1 X = x.reshape(-1, 4)
2 X
```

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

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

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

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

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

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

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

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

```
tensor([[ 0.5300,  0.1967, -0.0473, -0.7737],
        [-0.6123, -0.3709,  0.2209, -0.1136],
        [-0.7613, -0.0198, -0.2387,  0.3496]])
```

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

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

```
1 X
```

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

2.1.2. Index Slicing

- we can access tensor elements by indexing (starting with 0).
- includes the first index (start) but not the last (stop).

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

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

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

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

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

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

2.1.3. Elementwise Operations

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

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

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

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

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

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

```
1 X == Y
```

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

```
1 X.sum()
```

```
tensor(66.)
```

2.1.4. Broadcasting

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

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

```
1 a + b
```

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

2.1.5. Saving Memory

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

```
False
```

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

```
id(Z): 134823526854032
id(Z): 134823526854032
<function print>
```

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

```
True
```

2.1.6. Conversion to Other Python Objects

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

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

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

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

2.2. Data Preprocessing

2.2.1. Reading the Dataset

```
import os

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

```
import pandas as pd

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

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

2.2.2. Data Preparation

- first step in processing the dataset is to separate out columns corresponding to input versus target values. We can select columns either by name or via integer-location based indexing (iloc).
- missing values might be handled either via imputation or deletion.
- Imputation replaces missing values with estimates of their values while deletion simply discards either those rows or those columns that contain missing values.

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

```
3 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
```

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

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

2.2.3. Conversion to the Tensor Format

```
1 import torch
2 X = torch.tensor(inputs.to_numpy(dtype = float))
3 y = torch.tensor(targets.to_numpy(dtype=float))
4 X, y
```

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

2.3. Linear Algebra

```
1 import torch
```

2.3.1. Scalars

- We denote scalars by ordinary lower-cased letters (e.g.x,y and z) and the space of all (continuous) real-valued scalars by $\mathbb{R}$.
- Scalars are implemented as tensors that contain only one element.
- addition, multiplication, division, and exponentiation operations.

```
1 x = torch.tensor(3.0)
2 y = torch.tensor(2.0)
3 x + y, x * y, x / y, x ** y
```

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

2.3.2. Vectors

- we call these scalars the elements of the vector (synonyms include entries and components).
- When vectors represent examples from real-world datasets, their values hold some real-world significance.
- Vectors are implemented as 1st-order tensors.

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

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

```
1 x[2]
```

```
tensor(2)
```

```
1 len(x)
```

```
3
```

```
1 x.shape
```

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

2.3.3. Matrices

- Just as scalars are 0th-order tensors and vectors are 1st-order tensors, matrices are 2nd-order tensors.
- m X n real-valued scalars, arranged as rows and columns.

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

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

```
A.T
```

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

2.3.4. Tensors

- Tensors give us a generic way of describing extensions to nth-order arrays.
- Tensors will become more important when we start working with images. Each image arrives as a 3rd-order tensor with axes corresponding to the height, width, and channel.

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

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

2.3.5. Basic Properties of Tensor Arithmetic

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

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

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

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

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

```
A * B
```

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

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

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

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

2.3.6. Reduction

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

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

```
A.shape, A.sum()
```

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

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

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

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

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

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

```
tensor(True)
```

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

```
(tensor(2.5000), tensor(2.5000))
```

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

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

2.3.7. Non-Reduction Sum

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

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

```
A / sum_A
```

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

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

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

2.3.8. Dot Products

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

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

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

```
tensor(3.)
```

2.3.9. Matrix-Vector Products

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

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

2.3.10. Matrix-Matrix Multiplication

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

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

2.3.11. Norms

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

```
tensor(5.)
```

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

```
tensor(7.)
```

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

```
tensor(6.)
```

2.5. Automatic Differentiation

```
import torch
```

2.5.1. A Simple Function

- In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory.
- Note that the gradient of a scalar-valued function with respect to a vector x is vector-valued with the same shape as x. .

```
x = torch.arange(4.0)
x
```

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

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

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

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

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

```
x.grad == 4 * x
```

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

2.5.1.1. reset gradient

- To reset the gradient buffer, we can call x.grad.zero_() as follows

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

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

2.5.2. Backward for Non-Scalar Variables

- When y is a vector, the most natural representation of the derivative of y with respect to a vector x is a matrix called the Jacobian that contains the partial derivatives of each component of y with respect to each component of x.
- While Jacobians do show up in some advanced machine learning techniques, more commonly we want to sum up the gradients of each component of y with respect to the full vector x, yielding a vector of the same shape as x.

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

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

2.5.3. Detaching Computation

- we wish to move some calculations outside of the recorded computational graph.
- we need to detach the respective computational graph from the final result.

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

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

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

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

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

2.5.4. Gradients and Python Control Flow

- One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable.

```
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
```

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

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

```
tensor(True)
```

3.1 Linear Regression

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

3.1.1. Basics

- Linear regression is both the simplest and most popular among the standard tools for tackling regression problems.

3.1.1.1. Model

- The assumption of linearity means that the expected value of the target (price) can be expressed as a weighted sum of the features (area and age)
- y = w1x1 + ... + wdxd + b.
- $y = w^T x + b$

3.1.1.2. Loss Function

- Loss functions quantify the distance between the real and predicted values of the target. The loss will usually be a nonnegative number where smaller values are better and perfect predictions incur a loss of 0.
- Note that large differences between estimates and targets lead to even larger contributions to the loss.

3.1.1.3. Analytic Solution

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

3.1.1.4. Minibatch Stochastic Gradient Descent

- The key technique for optimizing nearly every deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called gradient descent.
- Minibatch SGD proceeds as follows: (i) initialize the values of the model parameters, typically at random; (ii) iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient.

3.1.2. Vectorization for Speed

- When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.
- Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the mathematics to the library so we do not have to write as many calculations ourselves, reducing the potential for errors and increasing portability of the code.

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

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

```
'0.10869 sec'
```

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

```
'0.00054 sec'
```

3.1.3. The Normal Distribution and Squared Loss

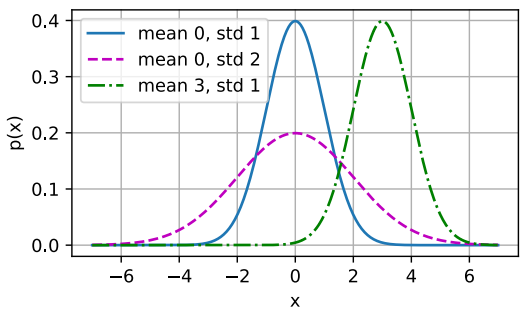
- changing the mean corresponds to a shift along the x-axis, and increasing the variance spreads the distribution out, lowering its peak.

```
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)
```

```
# 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])
```



3.2. Object-Oriented Design for Implementation

1. Module contains models, losses, and optimization methods
2. DataModule provides data loaders for training and validation
3. both classes are combined using the Trainer class, which allows us to train models on a variety of hardware platforms.

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

3.2.1. Utilities

- The first utility function allows us to register functions as methods in a class after the class has been created.
- we can do so even after we have created instances of the class! It allows us to split the implementation of a class into multiple code blocks.

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

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

a = A()
```

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

a.do()
```

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

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
# 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
```

3.2.1.1. draw method

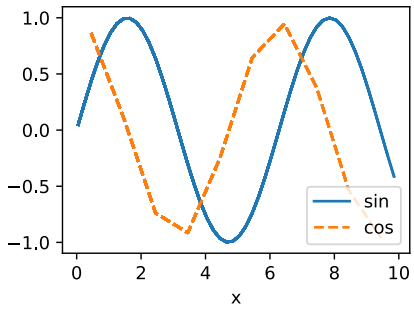
- The draw method plots a point (x, y) in the figure, with label specified in the legend. The optional every_n smooths the line by only showing points in the figure. Their values are averaged from the neighbor points in the original figure.

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

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

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



3.2.2. Models

- The first, **init**, stores the learnable parameters, the **training_step** method accepts a data batch to return the loss value, and finally, **configure_optimizers** returns the optimization method, or a list of them, that is used to update the learnable parameters.
- Optionally we can define **validation_step** to report the evaluation measures. Sometimes we put the code for computing the output into a separate **forward** method to make it more reusable.

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

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

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

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / W
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / W
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / W
                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)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
        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

- The DataModule class is the base class for data.
- Quite frequently the **init** method is used to prepare the data.
- The **train_dataloader** returns the data loader for the training dataset.
- This batch is then fed into the **training_step** method of Module to compute the loss. There is an optional **val_dataloader** to return the validation dataset loader. It behaves in the same manner, except that it yields data batches for the **validation_step** method in Module.

```
class DataModule(d2l.HyperParameters):  #@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)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.2.4. Training

- The **Trainer** class trains the learnable parameters in the Module class with data specified in **DataModule**.
- The key method is **fit**, which accepts two arguments: **model**, an instance of **Module**, and **data**, an instance of **DataModule**.
- It then iterates over the entire dataset max_epochs times to train the model.

```
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

    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
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.4. Linear Regression Implementation from Scratch

- we will implement the entire method from scratch

1. the model

2. the loss function
3. a minibatch stochastic gradient descent optimizer
4. the training function that stitches all of these pieces together.

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

3.4.1. Defining the Model

- we initialize weights by drawing random numbers from a normal distribution with mean 0 and a standard deviation of 0.01.
- Moreover we set the bias to 0.
- we simply take the matrix–vector product of the input features X and the model weights w, and add the offset b to each example.
- The product Xw is a vector and b is a scalar.

```
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=
        self.b = torch.zeros(1, requires_grad=True)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.4.2. Defining the Loss Function

- we need to transform the true value y into the predicted value's shape y_hat.
- The result returned by the following method will also have the same shape as y_hat.
- We also return the averaged loss value among all examples in the minibatch.

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.4.3. Defining the Optimization Algorithm

- At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of the loss with respect to the parameters. Next, we update the parameters in the direction that may reduce the loss.
- The following code applies the update, given a set of parameters, a learning rate lr. Since our loss is computed as an average over the minibatch, we do not need to adjust the learning rate against the batch size.

```
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_()
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.4.4. Exercise(Training)

- In each epoch, we iterate through the entire training dataset, passing once through every example (assuming that the number of examples is divisible by the batch size).

- In each iteration, we grab a minibatch of training examples, and compute its loss through the model's **training_step** method. Then we compute the gradients with respect to each parameter.
- Finally, we will call the optimization algorithm to update the model parameters.

```
@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
```

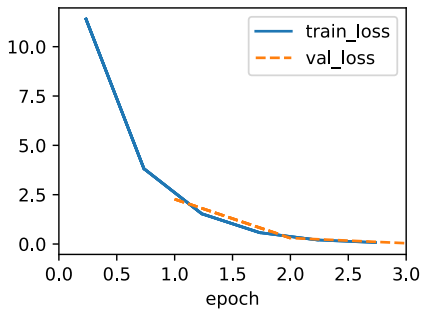
"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.4.4.1. Training data

- we need some training data. we use the SyntheticRegressionData class and pass in some ground truth parameters.
- we train our model with the learning rate lr=0.03 and set max_epochs=3, both the number of epochs and the learning rate are hyperparameters.

```
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)
```



3.4.4.2. Estimate

```
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([ 0.1143, -0.1437])
error in estimating b: tensor([0.2195])
```

4.1 Softmax Regression

- sort of time-to-event modeling comes with a host of other complications that are dealt with in a specialized subfield called *survival modeling*.
- The point here is that there is a lot more to estimation than simply minimizing squared errors. more broadly, supervised learning than regression.
- we focus on classification problems where we put aside how much? questions and instead focus on which category? questions.

4.1.1. Classification

- We have two obvious choices.
- Perhaps the most natural impulse would be to choose {1, 2, 3}, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer.
- *one-hot encoding* is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0.
    - In our case, a label would be a three-dimensional vector, with (1, 0, ,0) corresponding to "cat", (0, 1, 0) to "chicken", and (0, 0, 1) to "dog"

4.1.1.1. Linear Model

- In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class.
- To address classification with linear models, we will need as many affine functions as we have outputs. Strictly speaking, we only need one fewer, since the final category has to be the difference between 1 and the sum of the other categories, but for reasons of symmetry we use a slightly redundant parametrization.
- o = Wx + b

4.1.1.2. The Softmax

- There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.
- **Probit model**
    - we could assume that the outputs o are corrupted versions of y, where the corruption occurs by means of adding noise ϵ drawn from a normal distribution.
- **Nomalization**
    - use exponential function
- ***Softmax fuction***
    - y = softmax(o)
    - The idea of a softmax dates back to Gibbs (1902), who adapted ideas from physics.
    - Following Gibbs' idea, energy equates to error. Energy-based models (Ranzato et al., 2007) use this point of view when describing problems in deep learning.

4.1.1.3. Vectorization

- To improve computational efficiency, we vectorize calculations in minibatches of data.

4.1.2. Loss Function

- we need a way to optimize the accuracy of this mapping.
- We will rely on maximum likelihood estimation, the very same method that we encountered when providing a probabilistic justification for the mean squared error loss in Section 3.1.3.

4.1.2.1. Log-Likelihood

- We can compare the estimates with reality by checking how probable the actual classes are according to our model.
- We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution.
- Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood
- Even if our model could assign an output probability of 0, any error made when assigning such high confidence would incur infinite loss.

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

4.1.2.2. Softmax and Cross-Entropy Loss

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

- The derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector.

4.1.3. Information Theory Basics

- Information theory deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data).

4.1.3.1. Entropy

- The central idea in information theory is to quantify the amount of information contained in data.
- For a distribution P its entropy, H[P], is defined as:

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

- we need at least H[P] “nats” to encode it (Shannon, 1948).

4.1.3.2. Surprisal

$$\log \frac{1}{P(j)} = -\log P(j)$$

- Quantify one’s surprisal at observing an event j having assigned it a (subjective) probability P(j).
- The entropy defined in (4.1.11) is then the *expected surprisal* when one assigned the correct probabilities that truly match the data-generating process.

4.1.3.3. Cross-Entropy Revisited

- The cross-entropy from P to Q, denoted H(P, Q), is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according to probabilities P.

$$H(P, Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$$

1. as maximizing the likelihood of the observed data
2. as minimizing our surprisal (and thus the number of bits) required to communicate the labels.

4.2. The Image Classification Dataset

- MNIST serves as more of a sanity check than as a benchmark.

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

d2l.use_svg_display()
```

4.2.1. Loading the Dataset

- Since the Fashion-MNIST dataset is so useful, all major frameworks provide preprocessed versions of it.
- We can download and read it into memory using built-in framework utilities.
- Fashion-MNIST consists of images from 10 categories, each represented by 6000 images in the training dataset and by 1000 in the test dataset.
- A test dataset is used for evaluating model performance.
- By convention we store an image as a $c \times h \times w$ tensor, where c is the number of color channels, h is the height and w is the width.

```
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
        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
```

```
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-u
100%|██████████| 26421880/26421880 [00:02<00:00, 11259303.72it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw/train-labels-idx1-u
100%|██████████| 29515/29515 [00:00<00:00, 176978.77it/s]
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw/t10k-images-idx3-uby
100%|██████████| 4422102/4422102 [00:01<00:00, 3283170.86it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw/t10k-labels-idx1-uby
100%|██████████| 5148/5148 [00:00<00:00, 21065636.09it/s]Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/ra


(60000, 10000)
```

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

```
torch.Size([1, 32, 32])
```

```
@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]
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

4.2.2. Reading a Minibatch

- we use the built-in data iterator rather than creating one from scratch.
- shuffle the examples for the training data iterator.
- it is good enough that training a network will not be I/O constrained.

```
@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=t
                                       num_workers=self.num_workers)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

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

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

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

```
'10.33 sec'
```

4.2.3. Visualization

- **show_images** can be used to visualize the images and the associated labels.

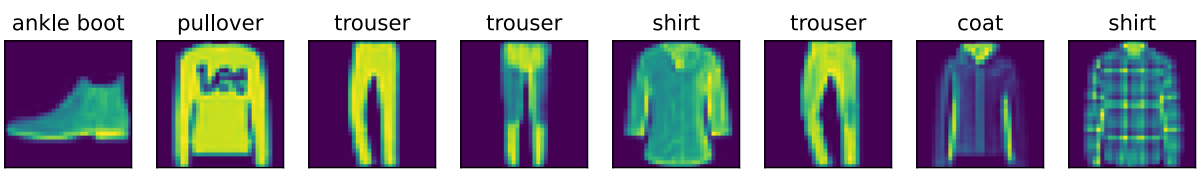
- Humans are very good at spotting oddities and because of that, visualization serves as an additional safeguard against mistakes and errors in the design of experiments.`

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

```
@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)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]



4.3. The Base Classification Model

- This section provides a base class for classification models to simplify future code.

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

4.3.1. The Classifier Class

- In the **validation_step** we report both the loss value and the classification accuracy on a validation batch.
- We draw an update for every **num_val_batches** batches.
- This has the benefit of generating the averaged loss and accuracy on the whole validation data.
- By default we use a stochastic gradient descent optimizer, operating on minibatches.

```
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)
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

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

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

4.3.2. Accuracy

- Given the predicted probability distribution **y_hat**, we typically choose the class with the highest predicted probability whenever we must output a hard prediction.
- The classification accuracy is the fraction of all predictions that are correct.

```
@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
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

4.4. Softmax Regression Implementation from Scratch

- we limit ourselves to defining the softmax-specific aspects of the model and reuse the other components from our linear regression section, including the training loop.

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

4.4.1. The Softmax

- Computing the softmax requires three steps
    1. exponentiation of each term
    2. a sum over each row to compute the normalization constant for each example
    3. division of each row by its normalization constant, ensuring that the result sums to 1:

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

- The denominator is called the (log) *partition function*.

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

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

```
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
```

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

```
(tensor([[0.1487, 0.2381, 0.1612, 0.2369, 0.2151],
         [0.1427, 0.2176, 0.1264, 0.2696, 0.2437]]),
 tensor([1., 1.]))
```

4.4.2. The Model

- In softmax regression, the number of outputs from our network should be equal to the number of classes.
- As with linear regression, we initialize the weights W with Gaussian noise. The biases are initialized as zeros.
- The code defines how the network maps each input to an output. Using **reshape** we flatten pixel image.

```
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]
```

```
@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

- Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label.
- We use indexing for efficiency.
- we can implement the cross-entropy loss function by averaging over the logarithms of the selected probabilities.

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

```
tensor([0.1000, 0.5000])
```

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

cross_entropy(y_hat, y)
```

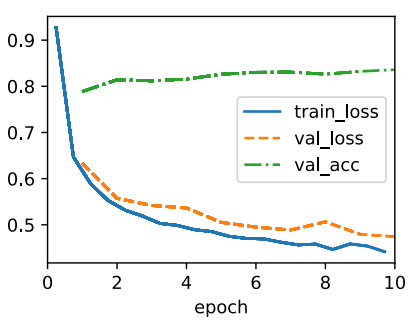
```
tensor(1.4979)
```

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

4.4.4. Training

- We reuse the fit method defined in Section 3.4 to train the model with 10 epochs.
- The number of epochs (max_epochs), the minibatch size (batch_size), and learning rate (lr) are adjustable hyperparameters.
- While these values are not learned during our primary training loop, they still influence the performance of our model, both vis-à-vis training and generalization performance.

```
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)
```



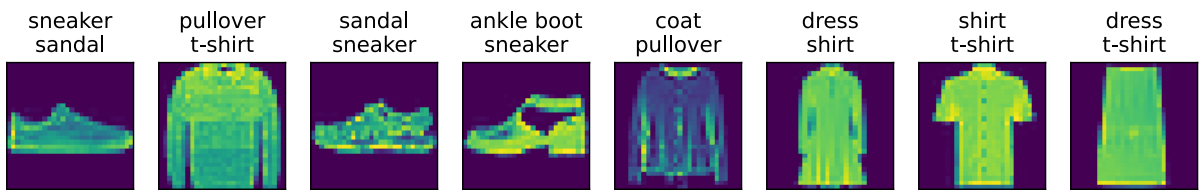
4.4.5. Prediction

- Now we can classify images.
- We are more interested in the images we label incorrectly. We visualize them by comparing their actual labels with the predictions from the model.

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

```
torch.Size([256])
```

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



5.1 Multilayer Perceptrons

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

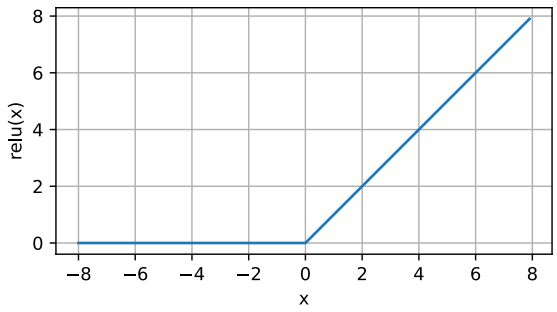
5.1.1.1. Memo

1. Universal Approximators

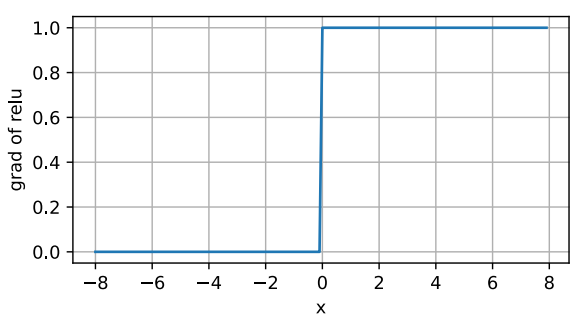
- A single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function.
- Kernel methods are way more effective, since they are capable of solving the problem exactly even in **infinite dimensional spaces** (Kimeldorf and Wahba, 1971, Schölkopf et al., 2001). " Universal Approximatation Theorem"
- We can approximate many functions much more **compactly** by using deeper (rather than wider) networks (Simonyan and Zisserman, 2014).

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

```
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))
```




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

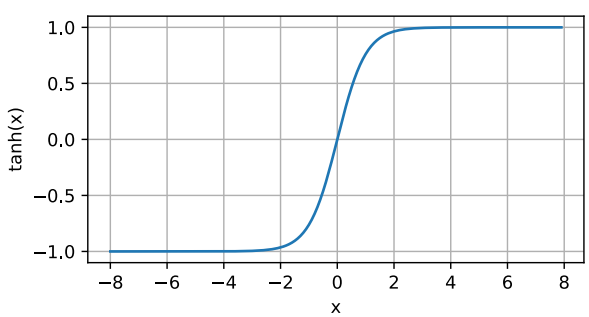


continued

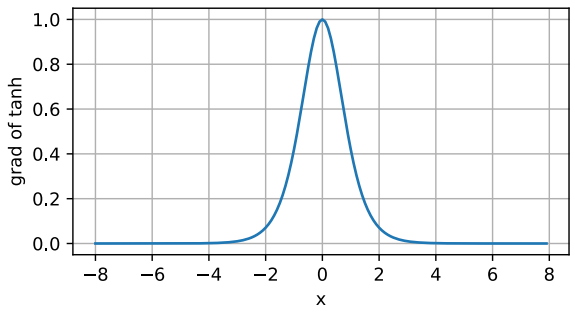
5.1.2.3 Tanh function (Discussion)

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

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



```
# 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 tanh', figsize=(5, 2.5))
```



5.2. Implementation of Multilayer Perceptrons

- Difference between MLP and simple linear models is we concatenate multiple layers.

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

5.2.1. Implementation from Scratch

5.2.1.1. Initializing Model Parameters

- To begin, we will implement an MLP with one hidden layer and 256 hidden units.
- For every layer, we must keep track of one weight matrix and one bias vector.
- **nn.Parameter** to automatically register a class attribute as a parameter to be tracked by **autograd**.

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

5.2.1.2. Model

- implementing the ReLU activation ourselves, we check everything works well.
- Since we are disregarding spatial structure, we **reshape** each two-dimensional image into a flat vector of length **num_inputs**.

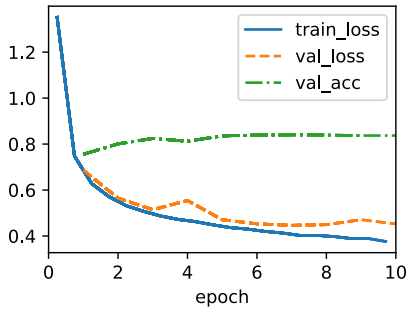
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

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

5.2.1.3. Training

- Same as for softmax regression.
- We define the model, data, and trainer, then finally invoke **the fit** method on model and data.

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



5.2.2. Concise Implementaion

5.2.2.1. Model

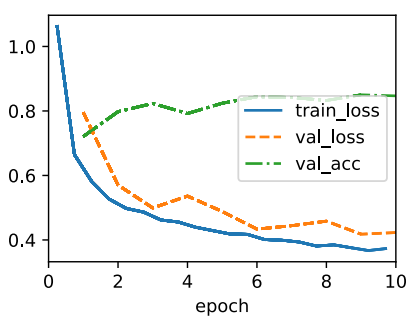
- We will add two fully connected layers.
    1. hidden layer
    2. output layer

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

5.2.2.2. Training

- Same as softmax regression.

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



5.3. Forward Propagation, Backward Propagation, and Computational Graphs

5.3.1. Forward Propagation

- *Forward propagation* refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$$
$$\mathbf{h} = \phi(\mathbf{z}).$$
$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$
$$L = l(\mathbf{o}, y).$$
$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_{\mathrm{F}}^2 + \|\mathbf{W}^{(2)}\|_{\mathrm{F}}^2\right),$$
$$J = L + s.$$

5.3.2. Computational Graph of Forward Propagation

- Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation.

- Squares denote variables and circles denote operators.
- The lower-left corner signifies the input and the upper-right corner is the output.
- The directions of the arrows are primarily rightward and upward.

5.3.3. Backpropagation

- Backpropagation refers to the method of calculating the gradient of neural network parameters.
- From output to the input layer.

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

- Let's apply.

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

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

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

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

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

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

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

5.3.4. Training Neural Networks

- When training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation.
- Training deeper networks using larger batch sizes more easily leads to out-of-memory errors.

Discussons & Exercises

2.1. Data Manipulation

2.1.1. torch.twos

```
1 torch.twoes((2, 3, 4)) # impossible
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-144-e22444c3446f> in <cell line: 1>()
----> 1 torch.twoes((2, 3, 4)) # impossible

/usr/local/lib/python3.10/dist-packages/torch/__init__.py in __getattr__(name)
   2214             return importlib.import_module(f".{name}", __name__)
   2215
-> 2216         raise AttributeError(f"module '{__name__}' has no attribute '{name}'")
   2217
   2218 def get_device_module(device: Optional[Union[torch.device, str]] = None):

AttributeError: module 'torch' has no attribute 'twoes'
```

다음 단계: 오류 설명

2.1.2. Memory discussion

- 일차원 벡터인 x를 reshape을 통해 2차원배열의 형태로 X에 저장했을 때 X의 값을 수정할 경우 x의 값도 같이 수정되는 것을 발견

```
x = torch.arange(12, dtype = torch.float32)
X = x.reshape(3, -1)
X[1, 1] = 0
print(x)
```

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

2.2. Data Preprocessing

2.2.2. Data Preparation

- first step in processing the dataset is to separate out columns corresponding to input versus target values. We can select columns either by name or via integer-location based indexing (iloc).
- missing values might be handled either via imputation or deletion.
- Imputation replaces missing values with estimates of their values while deletion simply discards either those rows or those columns that contain missing values.

2.2.3. loc

- iloc 대신 loc의 경우에도 csv파일을 추출할 수 있다. 대신 loc의 경우 행, 열의 인덱스번호를 사용하지 않고 key값을 통해 추출한다.

2.3. Linear Algebra

2.3.1. Scalars

- We denote scalars by ordinary lower-cased letters (e.g.x,y and z) and the space of all (continuous) real-valued scalars by $\mathbb{R}$.
- Scalars are implemented as tensors that contain only one element.
- addition, multiplication, division, and exponentiation operations.

2.3.2. Vectors

- we call these scalars the elements of the vector (synonyms include entries and components).
- When vectors represent examples from real-world datasets, their values hold some real-world significance.
- Vectors are implemented as 1st-order tensors.
- tensor에 len을 사용할 경우 3X2의 벡터일 때, 3이 출력된다. shape을 사용할 경우 [3, 2]라 출력된다.

[ ] ↳ *숨겨진 셀 4개*

2.3.3. Matrices

- Just as scalars are 0th-order tensors and vectors are 1st-order tensors, matrices are 2nd-order tensors.
- m X n real-valued scalars, arranged as rows and columns.

[ ] ↳ *숨겨진 셀 2개*

2.3.4. Tensors

- Tensors give us a generic way of describing extensions to nth-order arrays.
- Tensors will become more important when we start working with images. Each image arrives as a 3rd-order tensor with axes corresponding to the height, width, and channel.

[ ] ↳ *숨겨진 셀 1개*

2.3.5. Basic Properties of Tensor Arithmetic

[ ] ↳ *숨겨진 셀 4개*

2.3.6. Reduction

[ ] ↳ *숨겨진 셀 7개*

2.3.7. Non-Reduction Sum

[ ] ↳ *숨겨진 셀 3개*

2.3.8. Dot Products

[ ] ↳ *숨겨진 셀 3개*

2.3.9. Matrix–Vector Products

[ ] ↳ *숨겨진 셀 1개*

2.3.10. Matrix–Matrix Multiplication

[ ] ↳ *숨겨진 셀 1개*

2.3.11. Norms

[ ] ↳ *숨겨진 셀 3개*

2.5. Automatic Differentiation

2.5.1. A Simple Function

- In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory.
- Note that the gradient of a scalar-valued function with respect to a vector x is vector-valued with the same shape as x. .

[ ] ↳ *숨겨진 셀 7개*

2.5.2. Backward for Non-Scalar Variables

- When y is a vector, the most natural representation of the derivative of y with respect to a vector x is a matrix called the Jacobian that contains the partial derivatives of each component of y with respect to each component of x.
- While Jacobians do show up in some advanced machine learning techniques, more commonly we want to sum up the gradients of each component of y with respect to the full vector x, yielding a vector of the same shape as x.

[ ] ↳ *숨겨진 셀 1개*

2.5.3. Detaching Computation

- we wish to move some calculations outside of the recorded computational graph.
- we need to detach the respective computational graph from the final result.

[ ] ↳ *숨겨진 셀 2개*

2.5.4. Gradients and Python Control Flow

- One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable.

[ ] ↳ *숨겨진 셀 3개*

3.1. Linear Regression

- In the terminology of machine learning, the dataset is called a training dataset or training set, and each row (containing the data corresponding to one sale) is called an example (or data point, instance, sample). The thing we are trying to predict (price) is called a label (or target). The variables (age and area) upon which the predictions are based are called features (or covariates).

3.1.1. Basics

- Linear regression is both the simplest and most popular among the standard tools for tackling regression problems.

3.1.1.1. Model

- The assumption of linearity means that the expected value of the target (price) can be expressed as a weighted sum of the features (area and age)
- y = w1x1 + ... + wdxd + b.
- $y = w^T x + b$

3.1.1.2. Loss Function

- Loss functions quantify the distance between the real and predicted values of the target. The loss will usually be a nonnegative number where smaller values are better and perfect predictions incur a loss of 0.
- Note that large differences between estimates and targets lead to even larger contributions to the loss.

3.1.1.3. Analytic Solution

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

3.1.1.4. Minibatch Stochastic Gradient Descent

- The key technique for optimizing nearly every deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called gradient descent.
- Minibatch SGD proceeds as follows: (i) initialize the values of the model parameters, typically at random; (ii) iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient.

3.1.2. Vectorization for Speed

- When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.
- Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the mathematics to the library so we do not have to write as many calculations ourselves, reducing the potential for errors and increasing portability of the code.

[ ] ↳ *숨겨진 셀 3개*

3.1.3. The Normal Distribution and Squared Loss

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

[ ] ↳ *숨겨진 셀 2개*

3.2. Object-Oriented Design for Implementation

1. Module contains models, losses, and optimization methods
2. DataModule provides data loaders for training and validation
3. both classes are combined using the Trainer class, which allows us to train models on a variety of hardware platforms.

3.2.1. Utilities

- The first utility function allows us to register functions as methods in a class after the class has been created.
- we can do so even after we have created instances of the class! It allows us to split the implementation of a class into multiple code blocks.

3.2.1.1. draw method

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

[ ] ↳ *숨겨진 셀 2개*

3.2.2. Models

- The first, **init**, stores the learnable parameters, the **training_step** method accepts a data batch to return the loss value, and finally, **configure_optimizers** returns the optimization method, or a list of them, that is used to update the learnable parameters.
- Optionally we can define **validation_step** to report the evaluation measures. Sometimes we put the code for computing the output into a separate **forward** method to make it more reusable.

[ ] ↳ *숨겨진 셀 1개*

3.2.3. Data

- The DataModule class is the base class for data.
- Quite frequently the **init** method is used to prepare the data.
- The **train_dataloader** returns the data loader for the training dataset.
- This batch is then fed into the **training_step** method of Module to compute the loss. There is an optional **val_dataloader** to return the validation dataset loader. It behaves in the same manner, except that it yields data batches for the **validation_step** method in Module.

[ ] ↳ *숨겨진 셀 1개*

3.2.4. Training

- The **Trainer** class trains the learnable parameters in the Module class with data specified in **DataModule**.
- The key method is **fit**, which accepts two arguments: **model**, an instance of **Module**, and **data**, an instance of **DataModule**.
- It then iterates over the entire dataset max_epochs times to train the model.

[ ] ↳ *숨겨진 셀 1개*

3.4. Linear Regression Implementation from Scratch

- we will implement the entire method from scratch

1. the model
2. the loss function
3. a minibatch stochastic gradient descent optimizer
4. the training function that stitches all of these pieces together.

3.4.1. Defining the Model

- we initialize weights by drawing random numbers from a normal distribution with mean 0 and a standard deviation of 0.01.
- Moreover we set the bias to 0.
- we simply take the matrix–vector product of the input features X and the model weights w, and add the offset b to each example.
- The product Xw is a vector and b is a scalar.

[ ] ↳ *숨겨진 셀 2개*

3.4.2. Defining the Loss Function

- we need to transform the true value y into the predicted value's shape y_hat.
- The result returned by the following method will also have the same shape as y_hat.
- We also return the averaged loss value among all examples in the minibatch.

[ ] ↳ *숨겨진 셀 1개*

3.4.3. Defining the Optimization Algorithm

- At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of the loss with respect to the parameters. Next, we update the parameters in the direction that may reduce the loss.
- The following code applies the update, given a set of parameters, a learning rate lr. Since our loss is computed as an average over the minibatch, we do not need to adjust the learning rate against the batch size.

[ ] ↳ *숨겨진 셀 2개*

3.4.4. Exercise(Training)

- In each epoch, we iterate through the entire training dataset, passing once through every example (assuming that the number of examples is divisible by the batch size).
- In each iteration, we grab a minibatch of training examples, and compute its loss through the model's **training_step** method. Then we compute the gradients with respect to each parameter.
- Finally, we will call the optimization algorithm to update the model parameters.

```
@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
```

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

"@save" 주석은 허용되지 않습니다. 허용되는 값은 다음과 같습니다. [@param, @title, @markdown]

3.4.4.1. Training data

- we need some training data. we use the SyntheticRegressionData class and pass in some ground truth parameters.
- we train our model with the learning rate lr=0.03 and set max_epochs=3, both the number of epochs and the learning rate are hyperparameters.

```
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)
```

3.4.4.2. Estimate

```
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}')
```

4.1. Softmax Regression

- sort of time-to-event modeling comes with a host of other complications that are dealt with in a specialized subfield called *survival modeling*.
- The point here is that there is a lot more to estimation than simply minimizing squared errors. more broadly, supervised learning than regression.
- we focus on classification problems where we put aside how much? questions and instead focus on which category? questions.

4.1.1. Classification

- We have two obvious choices.
- Perhaps the most natural impulse would be to choose {1, 2, 3}, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer.
- *one-hot encoding* is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0.
    - In our case, a label would be a three-dimensional vector, with (1, 0, ,0) corresponding to "cat", (0, 1, 0) to "chicken", and (0, 0, 1) to "dog"

4.1.1.1. Linear Model

- In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class.
- To address classification with linear models, we will need as many affine functions as we have outputs. Strictly speaking, we only need one fewer, since the final category has to be the difference between 1 and the sum of the other categories, but for reasons of symmetry we use a slightly redundant parametrization.
- o = Wx + b

4.1.1.2. The Softmax

- There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.
- **Probit model**
    - we could assume that the outputs o are corrupted versions of y, where the corruption occurs by means of adding noise ϵ drawn from a normal distribution.
- **Nomalization**
    - use exponential function
- ***Softmax fuction***
    - y = softmax(o)
    - The idea of a softmax dates back to Gibbs (1902), who adapted ideas from physics.
    - Following Gibbs' idea, energy equates to error. Energy-based models (Ranzato et al., 2007) use this point of view when describing problems in deep learning.

4.1.1.3. Vectorization

- To improve computational efficiency, we vectorize calculations in minibatches of data.

4.1.2. Loss Function

- we need a way to optimize the accuracy of this mapping.
- We will rely on maximum likelihood estimation, the very same method that we encountered when providing a probabilistic justification for the mean squared error loss in Section 3.1.3.

4.1.2.1. Log-Likelihood

- We can compare the estimates with reality by checking how probable the actual classes are according to our model.
- We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution.
- Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood

- Even if our model could assign an output probability of 0, any error made when assigning such high confidence would incur infinite loss.

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

4.1.2.2. Softmax and Cross-Entropy Loss

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

- The derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector.

4.1.3. Information Theory Basics

- Information theory deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data).

4.1.3.1. Entropy

- The central idea in information theory is to quantify the amount of information contained in data.
- For a distribution P its entropy, H[P], is defined as:

$$H[P] = \sum_{j} -P(j) \log P(j).$$

- we need at least H[P] “nats” to encode it (Shannon, 1948).

4.1.3.2. Surprisal

$$\log \frac{1}{P(j)} = -\log P(j)$$

- Quantify one’s surprisal at observing an event j having assigned it a (subjective) probability P(j).
- The entropy defined in (4.1.11) is then the *expected surprisal* when one assigned the correct probabilities that truly match the data-generating process.

4.1.3.3. Cross-Entropy Revisited

- The cross-entropy from P to Q, denoted H(P, Q), is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according to probabilities P.

$$H(P, Q) \stackrel{\text{def}}{=} \sum_{j} -P(j) \log Q(j)$$

1. as maximizing the likelihood of the observed data
2. as minimizing our surprisal (and thus the number of bits) required to communicate the labels.

4.2. The Image Classification Dataset

- MNIST serves as more of a sanity check than as a benchmark.

4.2.1. Loading the Dataset

- Since the Fashion-MNIST dataset is so useful, all major frameworks provide preprocessed versions of it.
- We can download and read it into memory using built-in framework utilities.
- Fashion-MNIST consists of images from 10 categories, each represented by 6000 images in the training dataset and by 1000 in the test dataset.
- A test dataset is used for evaluating model performance.
- By convention we store an image as a $c \times h \times w$ tensor, where c is the number of color channels, h is the height and w is the width.

[ ] ↳ *숨겨진 셀 4개*

4.2.2. Reading a Minibatch

- we use the built-in data iterator rather than creating one from scratch.
- shuffle the examples for the training data iterator.

- it is good enough that training a network will not be I/O constrained.

[ ] ↳ *숨겨진 셀 3개*

4.2.3. Visualization

- **show_images** can be used to visualize the images and the associated labels.
- Humans are very good at spotting oddities and because of that, visualization serves as an additional safeguard against mistakes and errors in the design of experiments.`

[ ] ↳ *숨겨진 셀 2개*

4.3. The Base Classification Model

- This section provides a base class for classification models to simplify future code.

4.3.1. The Classifier Class

- In the **validation_step** we report both the loss value and the classification accuracy on a validation batch.
- We draw an update for every **num_val_batches** batches.
- This has the benefit of generating the averaged loss and accuracy on the whole validation data.
- By default we use a stochastic gradient descent optimizer, operating on minibatches.

[ ] ↳ *숨겨진 셀 2개*

4.3.2. Accuracy

- Given the predicted probability distribution **y_hat**, we typically choose the class with the highest predicted probability whenever we must output a hard prediction.
- The classification accuracy is the fraction of all predictions that are correct.

[ ] ↳ *숨겨진 셀 1개*

4.4. Softmax Regression Implementation from Scratch

- we limit ourselves to defining the softmax-specific aspects of the model and reuse the other components from our linear regression section, including the training loop.

4.4.1. The Softmax

- Computing the softmax requires three steps
    1. exponentiation of each term
    2. a sum over each row to compute the normalization constant for each example
    3. division of each row by its normalization constant, ensuring that the result sums to 1:

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

- The denominator is called the (log) *partition function*.

[ ] ↳ *숨겨진 셀 3개*

4.4.2. The Model

- In softmax regression, the number of outputs from our network should be equal to the number of classes.
- As with linear regression, we initialize the weights W with Gaussian noise. The biases are initialized as zeros.
- The code defines how the network maps each input to an output. Using **reshape** we flatten pixel image.

[ ] ↳ *숨겨진 셀 2개*

4.4.3. The Cross-Entropy Loss

- Recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true label.
- We use indexing for efficiency.
- we can implement the cross-entropy loss function by averaging over the logarithms of the selected probabilities.

[ ] ↳ *숨겨진 셀 3개*

4.4.4. Training

- We reuse the fit method defined in Section 3.4 to train the model with 10 epochs.
- The number of epochs (max_epochs), the minibatch size (batch_size), and learning rate (lr) are adjustable hyperparameters.
- While these values are not learned during our primary training loop, they still influence the performance of our model, both vis-à-vis training and generalization performance.

[ ] ↳ *숨겨진 셀 1개*

4.4.5. Prediction

- Now we can classify images.
- We are more interested in the images we label incorrectly. We visualize them by comparing their actual labels with the predictions from the model.

[ ] ↳ *숨겨진 셀 2개*

5.1 Multilayer Perceptrons

5.1.1.1. Memo

1. Universal Approximators

- A single-hidden-layer network, given enough nodes (possibly absurdly many), and the right set of weights, we can model any function.
- Kernel methods are way more effective, since they are capable of solving the problem exactly even in **infinite dimensional spaces** (Kimeldorf and Wahba, 1971, Schölkopf et al., 2001). " Universal Approximatation Theorem"
- We can approximate many functions much more **compactly** by using deeper (rather than wider) networks (Simonyan and Zisserman, 2014).

[ ] ↳ *숨겨진 셀 3개*

5.1.2.1. exercise

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

[ ] ↳ *숨겨진 셀 7개*

5.1.2.3 Tanh function (Discussion)

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

[ ] ↳ *숨겨진 셀 2개*

5.2. Implementation of Multilayer Perceptrons

- Difference between MLP and simple linear models is we concatenate multiple layers.

5.2.1. Implementation from Scratch

5.2.1.1. Initializing Model Parameters

- To begin, we will implement an MLP with one hidden layer and 256 hidden units.
- For every layer, we must keep track of one weight matrix and one bias vector.
- **nn.Parameter** to automatically register a class attribute as a parameter to be tracked by **autograd**.

[ ] ↳ *숨겨진 셀 1개*

5.2.1.2. Model

- implementing the ReLU activation ourselves, we check everything works well.
- Since we are disregarding spatial structure, we **reshape** each two-dimensional image into a flat vector of length **num_inputs**.

[ ] ↳ *숨겨진 셀 2개*

5.2.1.3. Training

- Same as for softmax regression.
- We define the model, data, and trainer, then finally invoke **the fit** method on model and data.

[ ] ↳ *숨겨진 셀 1개*

5.2.2. Concise Implementaion

5.2.2.1. Model

- We will add two fully connected layers.
  1. hidden layer
  2. output layer

[ ] ↳ *숨겨진 셀 1개*

5.2.2.2. Training

- Same as softmax regression.

[ ] ↳ *숨겨진 셀 1개*

5.3. Forward Propagation, Backward Propagation, and Computational Graphs

5.3.1. Forward Propagation

- *Forward propagation* refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

$$\begin{aligned}\mathbf{z} &= \mathbf{W}^{(1)}\mathbf{x}\\ \mathbf{h} &= \phi(\mathbf{z}).\\ \mathbf{o} &= \mathbf{W}^{(2)}\mathbf{h}.\\ L &= l(\mathbf{o}, y).\\ s &= \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_{\mathrm{F}}^2 + \|\mathbf{W}^{(2)}\|_{\mathrm{F}}^2\right),\\ J &= L + s.\end{aligned}$$

5.3.2. Computational Graph of Forward Propagation

- Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation.
- Squares denote variables and circles denote operators.
- The lower-left corner signifies the input and the upper-right corner is the output.
- The directions of the arrows are primarily rightward and upward.

5.3.3. Backpropagation

- Backpropagation refers to the method of calculating the gradient of neural network parameters.
- From output to the input layer.

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

- Let's apply.

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

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