COSE474-2024F Deep Learning HW 1

- **Chapter 2**
- 2021170964 박경빈

0.1. Installation

```
!pip install d2l==1.0.3
```

```
Requirement already satisfied: d2l==1.0.3 in /usr/local/lib/python3.10/dist-packages (1.0.3
Requirement already satisfied: jupyter==1.0.0 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: numpy==1.23.5 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: matplotlib==3.7.2 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: matplotlib-inline==0.1.6 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: requests==2.31.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: pandas==2.0.3 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: scipy==1.10.1 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-packages (from ju
Requirement already satisfied: qtconsole in /usr/local/lib/python3.10/dist-packages (from j
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-packages (from j
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-packages (from j
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-packages (from m
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from py
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/l
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from ju
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconv
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbco
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nb
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.10/dist-packages
```

2.1. Data Manipulation

2.1.1. Getting Started

```
import torch
```

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

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

```
x.numel()
```

```
12
```

```
x.shape
```

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

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

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

```
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.]]])
```

```
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.]]])
```

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

```
tensor([[-0.0206, -0.1437, -0.1006,  0.4550],
        [-0.4674, -2.3346,  0.7079,  0.5532],
        [ 1.3217, -0.7262,  1.0253, -0.3928]])
```

```
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]])
```

2.1.2. Indexing and Slicing

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

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

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

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

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

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

2.1.3. Operations

```
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])
```

```
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
```

```
(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.]))
```

```
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)
```

```
(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.]]))
```

```
X == Y
```

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

```
X.sum()
```

```
tensor(66.)
```

2.1.4. Broadcasting

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

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

```
# Without [:] -- Different ID
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140236852120768
id(Z): 140237176806336
```

```
# With [:] -- Same ID
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140236852127008
id(Z): 140236852127008
```

```
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

```
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
```

```
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
```

2.2.3. Conversion to the Tensor Format

```
import torch

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

2.3.1. Scalars

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

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

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

2.3.2. Vectors

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

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

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

```
x[2]
```

```
tensor(2)
```

```
len(x)
```

```
3
```

```
x.shape
```

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

2.3.3. Matrices

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

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

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

```
A.T
```

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

```
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]])
```

2.3.4. Tensors

```
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]]])
```

2.3.5. Basic Properties of Tensor Arithmetic

```
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.]]))
```

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}.$$

```
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()  # Same as 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

Euclidean norm:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}.$$

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

```
tensor(5.)
```

Manhattan distance:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n \left|x_i \right|.$$

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

```
tensor(7.)
```

Frobenius norm:

$$\|\mathbf{X}\|_\textrm{F} = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}.$$

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

```
tensor(6.)
```

2.5. Automatic Differentiation

2.5.1. A Simple Function

```
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])
```

```
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

```
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

```
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

```
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)
```

Discussions & Exercises

2.1. Discussion

Tensor

- Used for storing and manipulating data
- Popularly used in Deep Learning
- Provides indexing, slicing, mathematic operations, broadcasting, memory-efficient assignment, and conversion to/from python objects

2.1. Exercises & My Own Experiments

1. Run the code in this section. Change the conditional statement X == Y to X < Y or X > Y, and then see what kind of tensor you can get.
2. Replace the two tensors that operate by element in the broadcasting mechanism with other shapes, e.g., 3-dimensional tensors. Is the result the same as expected?

1. Run the code in this section. Change the conditional statement X == Y to X < Y or X > Y, and then see what kind of tensor you can get.

```
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]])
```

```
X
```

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

```
Y
```

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

```
X == Y
```

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

```
X < Y
```

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

```
X > Y
```

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

2. Replace the two tensors that operate by element in the broadcasting mechanism with other shapes, e.g., 3-dimensional tensors. Is the result the same as expected?

- Broadcasting doesn't work in 3 dimension if: Dimension with size of 1 Doesn't exist

```
a = torch.arange(12).reshape((2, 3, 2))
b = torch.arange(18).reshape((3, 2, 3))

print('a:', a)
print('b:', b)
print('a+b:', a+b)
```

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

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

        [[ 6,  7,  8],
         [ 9, 10, 11]],

        [[12, 13, 14],
         [15, 16, 17]]])
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-85-7e25581fd552> in <cell line: 8>()
      6 print('a:', a)
      7 print('b:', b)
----> 8 print('a+b:', a+b)

RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton
dimension 2
```

- Broadcasting doesn't work in 3 dimension if: None of the dimensions match each other
- In the example below, $2 \neq 3, 3 \neq 1, 1 \neq 3$, thus makes an error

```
a = torch.arange(6).reshape((2, 3, 1))
b = torch.arange(9).reshape((3, 1, 3))

print('a:', a)
print('b:', b)
print('a+b:', a+b)
```

```
a: tensor([[[0],
         [1],
         [2]],

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

        [[3, 4, 5]],

        [[6, 7, 8]]])
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-86-49f58a2d9551> in <cell line: 10>()
      8 print('a:', a)
      9 print('b:', b)
---> 10 print('a+b:', a+b)

RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-singleton
dimension 0
```

- Broadcasting works in 3 dimension if:

1. At least one dimesion match each other.
2. Two tensors must each own a dimension of size 1.

```
a = torch.arange(6).reshape((2, 3, 1))
b = torch.arange(6).reshape((2, 1, 3))
print('a:', a)
print('b:', b)
print('a+b:', a+b)
```

```
a: tensor([[[0],
         [1],
         [2]],

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

        [[3, 4, 5]]])
a+b: tensor([[[ 0,  1,  2],
         [ 1,  2,  3],
         [ 2,  3,  4]],

        [[ 6,  7,  8],
         [ 7,  8,  9],
         [ 8,  9, 10]]])
```

2.2. Discussion

Data Preprocessing might be challenging in real-world such as:

- Rather than arriving in a single CSV file, our dataset might be spread across **multiple files extracted from a relational database**, for example, e-commerce application
- Myriad data types might exist beyond just categorical and numeric, for example, **text strings, images, audio data, and point clouds**
- Advanced tools and efficient algorithms are required in order to prevent data processing from becoming the biggest bottleneck in the machine learning pipeline
- We must pay attention to data quality, since datasets are often **plagued by outliers**, **faulty measurements** from sensors, and **recording errors**, which must be addressed before feeding the data into any model

Data visualization tools such as **seaborn, Bokeh, or matplotlib** can help us to manually inspect the data

2.2. Exercises & My Own Experiments

1. Try loading datasets, e.g., Abalone from the UCI Machine Learning Repository and inspect their properties. What fraction of them has missing values? What fraction of the variables is numerical, categorical, or text?
2. Try indexing and selecting data columns by name rather than by column number. The pandas documentation on indexing has further details on how to do this.
3. How large a dataset do you think you could load this way? What might be the limitations? Hint: consider the time to read the data, representation, processing, and memory footprint. Try this out on your laptop. What happens if you try it out on a server?
4. How would you deal with data that has a very large number of categories? What if the category labels are all unique? Should you include the latter?
5. What alternatives to pandas can you think of? How about loading NumPy tensors from a file? Check out Pillow, the Python Imaging Library.

---

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

```
!pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.7-py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: certifi>=2020.12.5 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (fro
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from pyth
Downloading ucimlrepo-0.0.7-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.7
```

```python
from ucimlrepo import fetch_ucirepo

# fetch dataset
abalone = fetch_ucirepo(id=1)

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

# metadata
print(abalone.metadata)

# variable information
print(abalone.variables)
```

```
{'uci_id': 1, 'name': 'Abalone', 'repository_url': 'https://archive.ics.uci.edu/dataset/1/aba
             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
```

```python
import pandas as pd
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data"
data = pd.read_csv(url, header=None)
print(data)
```

```
      0      1      2      3       4       5       6       7   8
0     M  0.455  0.365  0.095  0.5140  0.2245  0.1010  0.1500  15
1     M  0.350  0.265  0.090  0.2255  0.0995  0.0485  0.0700   7
```

```
2     F  0.530  0.420  0.135  0.6770  0.2565  0.1415  0.2100   9
3     M  0.440  0.365  0.125  0.5160  0.2155  0.1140  0.1550  10
4     I  0.330  0.255  0.080  0.2050  0.0895  0.0395  0.0550   7
...  ..    ...    ...    ...     ...     ...     ...     ...  ..
4172  F  0.565  0.450  0.165  0.8870  0.3700  0.2390  0.2490  11
4173  M  0.590  0.440  0.135  0.9660  0.4390  0.2145  0.2605  10
4174  M  0.600  0.475  0.205  1.1760  0.5255  0.2875  0.3080   9
4175  F  0.625  0.485  0.150  1.0945  0.5310  0.2610  0.2960  10
4176  M  0.710  0.555  0.195  1.9485  0.9455  0.3765  0.4950  12

[4177 rows x 9 columns]
```

- **Properties:** Sex, Length, Diameter, Height, Whole_weight, Shucked_weight, Viscera_weight, Shell_weight, Rings
- None of them has any missing values
- **Categorial property:** Sex
- **Numerical property (Continuous):** Length, Diameter, Height, Whole_weight, Shucked_weight, Viscera_weight, Shell_weight
- **Numerical property (Integer):** Rings
- **Text property:** None

---

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

```
column_labels = ['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', 'Viscera
data.columns = column_labels
```

```
inputs, targets = data[['Sex', 'Length', 'Diameter', 'Height', 'Whole_weight', 'Shucked_weight', '
```

```
inputs
```

	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
...	...	...	...	...	...	...	...
4172	F	0.565	0.450	0.165	0.8870	0.3700	0.2390
4173	M	0.590	0.440	0.135	0.9660	0.4390	0.2145
4174	M	0.600	0.475	0.205	1.1760	0.5255	0.2875
4175	F	0.625	0.485	0.150	1.0945	0.5310	0.2610
4176	M	0.710	0.555	0.195	1.9485	0.9455	0.3765

4177 rows × 8 columns

```
targets
```

	Rings
0	15
1	7
2	9
3	10
4	7
...	...
4172	11
4173	10
4174	9
4175	10
4176	12

4177 rows × 1 columns

dtype: int64

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

My 16GB-RAM PC:

- Small to medium datasets up to a few GB should load quickly, but not for larger datasets
- My Estimation: A DataFrame with 8 columns might use about 100-200 bytes per row(depending on data types). With 16GB RAM, you might be able to load around 80-160 million rows before running out of memory.
- For datasets approaching the RAM limit, even simple operations could cause swapping, severely impacting performance.
- Background processes and the OS also need memory, further limiting available space.

A really good server (such as AWS):

- Servers often have faster I/O capabilities and much bigger memory, allowing quicker data loading, and handling much larger datasets more efficiently.
- With more RAM space, servers can handle datasets 8-64 times larger than a 16GB PC, which could work with billions of rows, depending on the exact specifications.
- Furthermore, better CPUs and GPU acceleration allow for faster processing of large datasets.
- Efficiently use of memory with techniques like memory-mapped files or distributed processing frameworks.

---

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

Yes, I would definitely use servers such as AWS to store & load a bunch of data safely & efficiently.

- Maybe *Amazon S3* for storing datas and files, or *Amazon SageMaker* due to its built-in algorithms and deep learning frameworks

---

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

NumPy tensors:

- NumPy arrays are more memory-efficient than pandas DataFrames for numerical data, and better for multi-dimensional tensors.
- However, it does not support alot of functionalities that are crutial for deep learning, so I might not use it for an alternative for pandas

Pillow:

- Pillow is known for its efficiency for loading, manipulating, and saving various image formats.
- However, Pillow is limited to 2D image data, which is not suitable for data manipulation or analysis.

Because of the importance of deep learning related tools / functionalities and ability to manipulate large dataframes, I would still use pandas for deep learning.

2.3. Discussion

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as sum and mean, respectively.
- Elementwise products are called Hadamard products. By contrast, dot products, matrix–vector products, and matrix–matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix–matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the l_1 and l_2 norms, and common matrix norms include the spectral and Frobenius norms.
- **The Frobenius norm behaves as if it were an l_2 norm of a matrix-shaped vector. Invoking the following function will calculate the Frobenius norm of a matrix.**

Frobenius norm:

$$\|\mathbf{X}\|_\mathrm{F} = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}.$$

2.3. Exercises & My Own Experiments

- **3 Examples of Frobenius Norm:**

```
torch.ones((5, 5))
```

```
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.norm(torch.ones((5, 5)))
```

```
tensor(5.)
```

```
torch.rand(3, 4)
```

```
tensor([[0.8744, 0.1753, 0.6514, 0.1203],
        [0.3977, 0.7149, 0.5240, 0.0659],
        [0.4611, 0.6816, 0.0527, 0.6883]])
```

```
torch.norm(torch.rand(3, 4))
```

```
tensor(1.8264)
```

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

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

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

```
tensor(5.4772)
```

2.5. Discussion

Simple Function's Automatic Differentiation

```
x = torch.arange(4.0)
y = 2 * torch.dot(x, x)
y.backward()
x.grad
x.grad == 4 * x
```

$$x = [x_1 \ , x_2 \ , x_3 \ , x_4 \ ]$$

$$y = 2 \cdot \mathrm{dot}(x, x) = 2(x_1^2 + x_2^2 + x_3^2 + x_4^2)$$

If we use `y.backward()`,

$$\frac{\partial y}{\partial x_1} = 2 \cdot 2x_1 = 4x_1$$

$$\frac{\partial y}{\partial x_2} = 2 \cdot 2x_2 = 4x_2$$

$$\frac{\partial y}{\partial x_3} = 2 \cdot 2x_3 = 4x_3$$

$$\frac{\partial y}{\partial x_4} = 2 \cdot 2x_4 = 4x_4$$

Thus, `x.grad()` is like following:

$$\nabla y = \left[\frac{\partial y}{\partial x_1} \ \frac{\partial y}{\partial x_2} \ \frac{\partial y}{\partial x_3} \ \frac{\partial y}{\partial x_4} \right] = [\, 4x_1 \ 4x_2 \ 4x_3 \ 4x_4 \,] = 4x$$

더블클릭 또는 Enter 키를 눌러 수정

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

$$x = [x_1, x_2, x_3, x_4]$$

$$y = \sum_{i=1}^{4} x_i = x_1 + x_2 + x_3 + x_4$$

If we use `y.backward()`,

$$\frac{\partial y}{\partial x_1} = 1$$

$$\frac{\partial y}{\partial x_2} = 1$$

$$\frac{\partial y}{\partial x_3} = 1$$

$$\frac{\partial y}{\partial x_4} = 1$$

Thus, `x.grad` is like following:

$$\nabla y = \left[\frac{\partial y}{\partial x_1} \ \frac{\partial y}{\partial x_2} \ \frac{\partial y}{\partial x_3} \ \frac{\partial y}{\partial x_4} \right] = [\, 1 \ 1 \ 1 \ 1 \,] = \mathbf{1}$$

Backward for Non-Scalar Variables

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

Why is `y.sum().backward()` faster than `y.backward(gradient=torch.ones(len(y)))`?

Case 1: `y.backward(gradient=torch.ones(len(y)))`

$$\frac{\partial y}{\partial x} = [2x_1, 2x_2, \ldots, 2x_n]$$

Here, y is a vector, and for each element $y_i = x_i^2$, we calculate:

$$\frac{\partial y_i}{\partial x_i} = 2x_i$$

and applying a multiplication with the gradient vector:

$$\mathbf{1} \cdot [2x_1, 2x_2, \ldots, 2x_n] = [2x_1, 2x_2, \ldots, 2x_n]$$

Case 2: `y.sum().backward()`

$$y_{\text{sum}} = \sum_{i=1}^{n} x_i^2$$

This gives a scalar value, and then you compute the gradient of this scalar with respect to x. The gradient of the sum with respect to each element x_i is:

$$\frac{\partial y_{\text{sum}}}{\partial x_i} = 2x_i$$

This is computed directly without needing to pass through each element of y.

Detaching Computation

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

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

$$y = [x_0^2, x_1^2, x_2^2, x_3^2] = [0^2, 1^2, 2^2, 3^2] = [0, 1, 4, 9]$$

which makes

$$u = [0, 1, 4, 9]$$

However, since `u = y.detach()` no gradient information is passed from `u` back to `y` or `x`.

$$z = u * x = [0 * 0, 1 * 1, 4 * 2, 9 * 3] = [0, 1, 8, 27]$$

$$z_{\text{sum}} = 0 + 1 + 8 + 27 = 36$$

Then, if we apply `backward()`,

$$\frac{\partial z_{\text{sum}}}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\sum_{j=0}^{3} u_j x_j\right)$$

where u_j **is considered constant**. Thus,

$$\frac{\partial z_{\text{sum}}}{\partial x_i} = u_i$$

$$\nabla z_{\text{sum}} = [u_0, u_1, u_2, u_3] = [0, 1, 4, 9]$$

2.5. Exercises & My Own Experiments

3. In the control flow example where we calculate the derivative of `d` with respect to `a`, what would happen if we changed the variable `a` to a random vector or a matrix? At this point, the result of the calculation `f(a)` is no longer a scalar. What happens to the result? How do we analyze this?
4. Let $f(x) = \sin(x)$. Plot the graph of f and of its derivative f'. Do not exploit the fact that $f'(x) = \cos(x)$ but rather use automatic differentiation to get the result.
5. Let $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$. Write out a dependency graph tracing results from x to $f(x)$.
6. Use the chain rule to compute the derivative df/dx of the aforementioned function, placing each term on the dependency graph that you constructed previously.

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

Gradients and Python Control Flow

```
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
```

Let's assume the loop iterates n times such that $b = a \cdot 2^{n+1}$ after the loop finishes.

Case 1: b.sum() > 0

$$c = a \cdot 2^{n+1}$$
$$\frac{\partial c}{\partial a} = 2^{n+1}$$
$$a.grad = \frac{\partial d}{\partial a} = 2^{n+1} = \frac{d}{a}$$

This is exactly what the expression a.grad == d / a represents.

Case 2: b.sum() <= 0

$$c = 100 \cdot a \cdot 2^{n+1}$$
$$\frac{\partial c}{\partial a} = 100 \cdot 2^{n+1}$$
$$a.grad = \frac{\partial d}{\partial a} = 100 \cdot 2^{n+1} = \frac{d}{a}$$

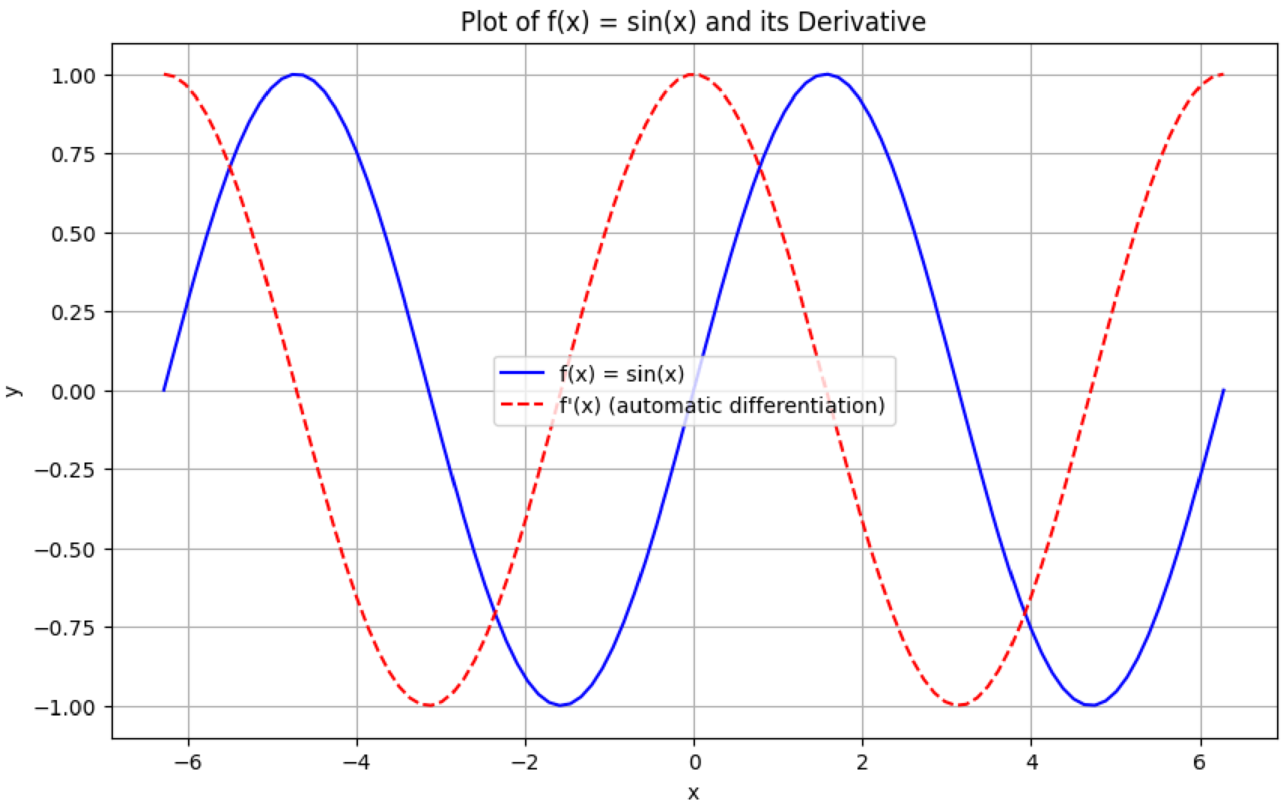
This also satisfies the expression a.grad == d / a.

---

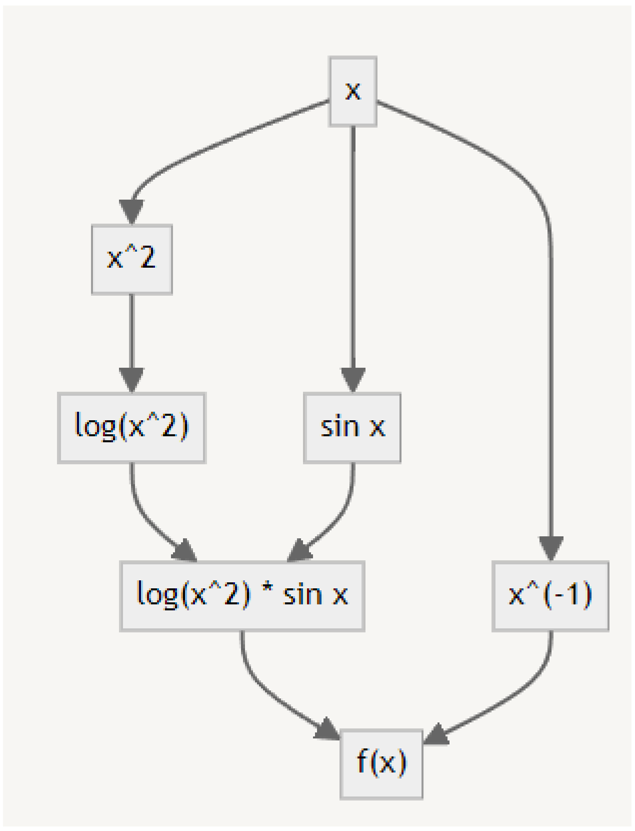
4. Let $f(x) = \sin(x)$. Plot the graph of f and of its derivative f'. Do not exploit the fact that $f'(x) = \cos(x)$ but rather use automatic differentiation to get the result.

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return torch.sin(x)

x_values = torch.linspace(-2 * np.pi, 2 * np.pi, 100, requires_grad=True)
y_values = f(x_values)

y_values.sum().backward()
grad_values = x_values.grad
```

```
x_np = x_values.detach().numpy()
y_np = y_values.detach().numpy()
grad_np = grad_values.detach().numpy()

plt.figure(figsize=(10, 6))
plt.plot(x_np, y_np, label="f(x) = sin(x)", color='blue')
plt.plot(x_np, grad_np, label="f'(x) (automatic differentiation)", color='red', linestyle='--')

plt.title("Plot of f(x) = sin(x) and its Derivative")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()

plt.grid(True)
plt.show()
```



5. Let $f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$. Write out a dependency graph tracing results from x to $f(x)$.



6. Use the chain rule to compute the derivative df/dx of the aforementioned function, placing each term on the dependency graph that you constructed previously

$$f(x) = ((\log x^2) \cdot \sin x) + x^{-1}$$

The function has two main components:

$$f_1(x) = (\log x^2) \cdot \sin x$$
$$f_2(x) = x^{-1}$$

To find derivative of $f_1(x) = (\log x^2) \cdot \sin x$ lets say:

$$g(x) = \log x^2$$
$$h(x) = \sin x$$

$$\frac{d}{dx}(g(x) \cdot h(x)) = g'(x) \cdot h(x) + g(x) \cdot h'(x)$$

$$g'(x) = \frac{d}{dx}(2 \log x) = \frac{2}{x}$$

$$h'(x) = \cos x$$

$$\frac{d}{dx}\left((\log x^2) \cdot \sin x\right) = \frac{2}{x} \cdot \sin x + (\log x^2) \cdot \cos x$$

To find derivative $f_2(x) = x^{-1}$,

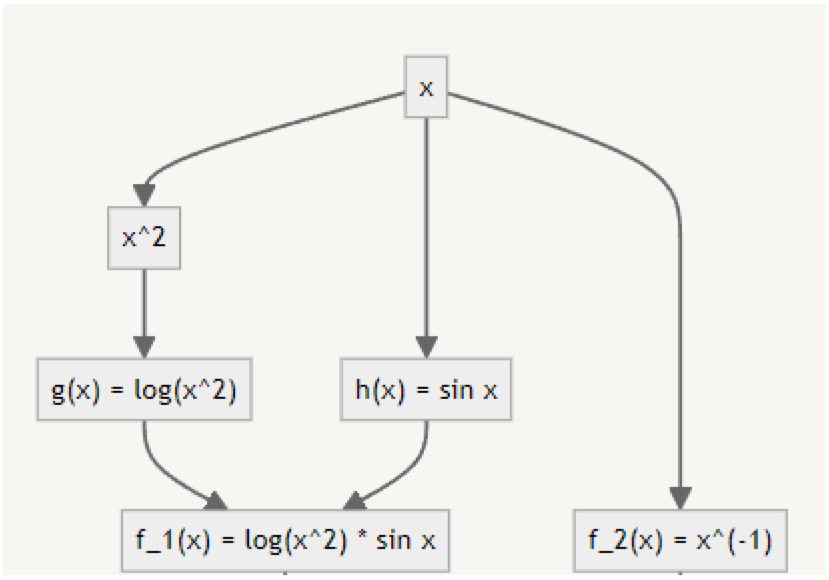
$$\frac{d}{dx}\left(x^{-1}\right) = -x^{-2} = -\frac{1}{x^2}$$

Thus, the final result:

$$\frac{df}{dx} = \frac{2}{x} \cdot \sin x + (\log x^2) \cdot \cos x - \frac{1}{x^2}$$

.

Which is just the opposite path of the below image:



COSE474-2024F Deep Learning HW 1

- **Chapter 3**
- 2021170964 박경빈

0.1. Installation

[ ] ↳ 숨겨진 셀 2개

3.1. Linear Regression

3.1.1. Basics

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

3.1.1.1. Model

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b.$$

Here w_{area} and w_{age} are called *weights*, and b is called a *bias* (or *offset* or *intercept*). The weights determine the influence of each feature on our prediction. The bias determines the value of the estimate when all features are zero.

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b.$$

(in general the "hat" symbol denotes an estimate)

Collecting all features into a vector $\mathbf{x} \in \mathbb{R}^d$ and all weights into a vector $\mathbf{w} \in \mathbb{R}^d$,

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

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.
- The most common loss function is the squared error

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)}\right)^2.$$

To measure the quality of a model on the entire dataset of n examples, we simply average (or equivalently, sum) the losses on the training set:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2.$$

When training the model, we seek parameters ($\mathbf{w}^*, b^*$) that minimize the total loss across all training examples:

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b).$$

3.1.1.3. Analytic Solution

Linear regression presents us with a surprisingly easy optimization problem. We can find the optimal parameters analytically by applying a simple formula as follows.

- First, we can subsume the bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all 1s.
- Then our prediction problem is to minimize $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$. As long as the design matrix $\mathbf{X}$ has full rank (no feature is linearly dependent on the others), then there will be just one critical point on the loss surface and it corresponds to the minimum of the loss over the entire domain.
- Taking the derivative of the loss with respect to $\mathbf{w}$ and setting it equal to zero yields:

$$\partial_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \text{ and hence } \mathbf{X}^\top \mathbf{y} = \mathbf{X}^\top \mathbf{X}\mathbf{w}.$$

Solving for $\mathbf{w}$ provides us with the optimal solution

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$$

which only is unique when the matrix $\mathbf{X}^\top\mathbf{X}$ is invertible (=when the columns of the design matrix are linearly independent)

3.1.1.4. Minibatch Stochastic Gradient Descent

Minibatch Stochastic Gradient Descent (SGD) is an optimization technique used to iteratively reduce error in deep learning models by updating model parameters.

- Instead of using the entire dataset (as in traditional gradient descent) or a single example (as in stochastic gradient descent), minibatch SGD updates the model based on a small random subset of the data (typically 32-256 samples).
- This balances efficiency and statistical robustness, leveraging faster computations like matrix--vector multiplications while avoiding the limitations of using a full dataset or individual samples.

In its most basic form, in each iteration t,

- randomly sample a minibatch $\mathcal{B}_t$ consisting of a fixed number $|\mathcal{B}|$ of training examples.
- compute the derivative (gradient) of the average loss on the minibatch with respect to the model parameters.
- Finally, we multiply the gradient by a predetermined small positive value η, called the *learning rate*, and subtract the resulting term from the current parameter values. We can express the update as follows:

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b).$$

In summary, minibatch SGD proceeds as follows:

1. initialize the values of the model parameters, typically at random;
2. iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient. For quadratic losses and affine transformations, this has a closed-form expansion:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)$$

In **minibatch stochastic gradient descent**, the minibatch size and learning rate are considered *hyperparameters*, which are set before training and not updated during the process. These can be optimized using techniques like **Bayesian optimization**.

- After training for a set number of iterations, the learned model parameters ($\hat{\mathbf{w}}, \hat{b}$) are recorded, though they may not be exact minimizers due to randomness in minibatch selection and the nature of deep learning loss surfaces, which often contain many minima and saddle points.
- The goal is not to find the exact parameters but ones that generalize well to unseen data, a key challenge known as *generalization*.

3.1.1.5. Predictions

Given the model $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, we can now make *predictions* for a new example, e.g., predicting the sales price of a previously unseen house given its area x_1 and age x_2.

3.1.2. Vectorization for Speed

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

First Method:

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

```
'0.19830 sec'
```

Second Method:

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

```
'0.00022 sec'
```

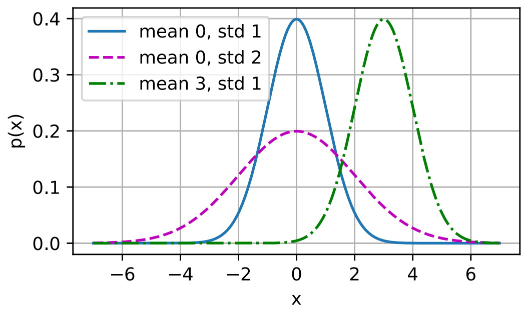
The second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the mathematics to the library so we do not have to write as many calculations ourselves, reducing the potential for errors and increasing portability of the code.

3.1.3. The Normal Distribution and Squared Loss

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

```
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])
```



Changing the mean of a distribution shifts it along the x-axis, while increasing the variance spreads it out, lowering its peak.

To motivate linear regression with squared loss, assume observations come from noisy measurements, where noise ϵ follows a normal distribution: $\mathcal{N}(0, \sigma^2)$. The likelihood of observing a particular y for a given $\mathbf{x}$ is:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2).$$

Thus, we can now write out the *likelihood* of seeing a particular y for a given $\mathbf{x}$ via

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right).$$

As such, the likelihood factorizes. According to *the principle of maximum likelihood*, the best values of parameters $\mathbf{w}$ and b are those that **maximize the likelihood** of the entire dataset:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}).$$

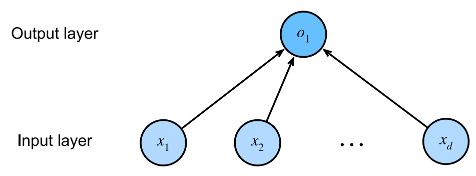
The equality follows since all pairs $(\mathbf{x}^{(i)}, y^{(i)})$ were drawn independently of each other. Estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*.

- For historical reasons, optimizations are more often expressed as minimization rather than maximization. So, without changing anything, we can *minimize* the *negative log-likelihood*, which we can express as follows:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2.$$

If we assume that σ is fixed, we can ignore the first term, because it does not depend on $\mathbf{w}$ or b. The second term is identical to the squared error loss introduced earlier, except for the multiplicative constant $\frac{1}{\sigma^2}$. Fortunately, the solution does not depend on σ either. It follows that minimizing the mean squared error is equivalent to the maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

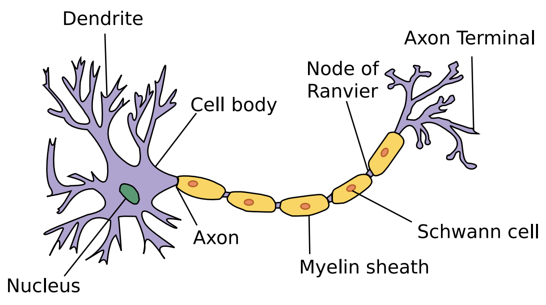
3.1.4. Linear Regression as a Neural Network



The inputs are $x_1, \ldots, x_d$. We refer to d as the *number of inputs* or the *feature dimensionality* in the input layer. The output of the network is o_1. Because we are just trying to predict a single numerical value, we have only one output neuron. Note that the input values are all *given*. There is just a single *computed* neuron. In summary, we can think of linear regression as a single-layer fully connected neural network. We will encounter networks with far more layers in later chapters.

3.1.4.1. Biology

Though linear regression predates computational neuroscience, it was a natural starting point for early models of artificial neurons developed by McCulloch and Pitts. They drew inspiration from biological neurons, with dendrites for inputs, a nucleus as the processor, and axons for outputs, connecting to other neurons via synapses.



:label: `fig_Neuron`

- Information x_i arriving from other neurons (or environmental sensors) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i, determining the effect of the inputs, e.g., activation or inhibition via the product $x_i w_i$.
- The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$, possibly subject to some nonlinear postprocessing via a function $\sigma(y)$.
- This information is then sent via the axon to the axon terminals, where it reaches its destination (e.g., an actuator such as a muscle) or it is fed into another neuron via its dendrites.

3.2. Object-Oriented Design for Implementation

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

3.2.1. Utilities

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

```
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
```

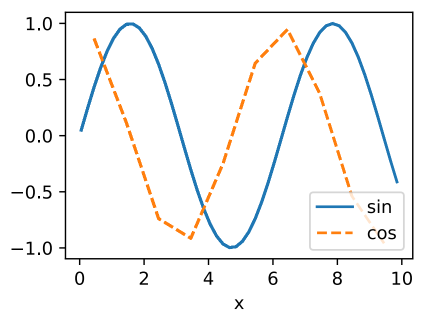
```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

```
# 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
```

```
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
```

```
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

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

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

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

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / ₩
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / ₩
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / ₩
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        raise NotImplementedError
```

3.2.3. Data

```
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)
```

3.2.4. Training

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

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

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

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

    def fit_epoch(self):
        raise NotImplementedError
```

3.4. Linear Regression Implementation from Scratch

```
%matplotlib inline
import torch
```

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

3.4.1. Defining the Model

```
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)
```

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

3.4.2. Defining the Loss Function

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

3.4.3. Defining the Optimization Algorithm

```
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_()
```

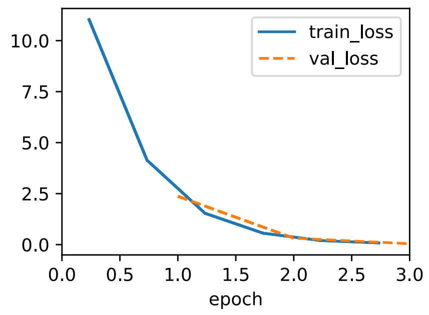
```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

3.4.4. Training

```
@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:  # 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
```

```
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)
```



```
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([ 0.1268, -0.1480])
error in estimating b: tensor([0.2158])
```

Discussions & Exercises

3.1. Discussion

- We explored traditional linear regression, where the parameters of a linear function are chosen to minimize squared loss on the training set.
- This approach was motivated by both practical considerations and its interpretation as **maximum likelihood estimation** under the assumption of *linearity* and *Gaussian noise*.
- After discussing both computational considerations and connections to statistics, we showed how such linear models could be expressed as simple neural networks where the inputs are directly wired to the output(s).
- They are sufficient to introduce most of the components that all of our models require: *parametric forms, differentiable objectives, optimization* via **minibatch stochastic gradient descent**, and ultimately, *evaluation* on previously unseen data.

3.2. Discussion

We used OOD (Object-Oriented Design) to implement a trainable model:

```
class Module(nn.Module, d2l.HyperParameters)
```

```
class DataModule(d2l.HyperParameters)
```

```
class Trainer(d2l.HyperParameters)
```

Moreover, these fully implemented classes are saved in the D2L library, a lightweight toolkit that makes structured modeling for deep learning easy.

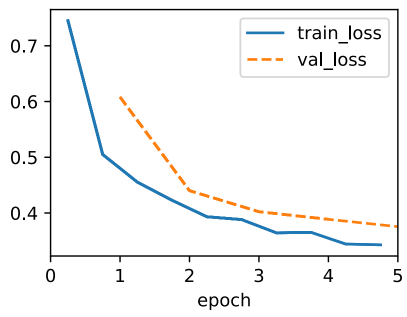
- We can reuse many components between projects without changing much at all.
- We can replace just the optimizer, just the model, just the dataset, etc.; this degree of modularity pays dividends throughout the book in terms of conciseness and simplicity

3.2. Exercises & My Own Experiments

1. Locate full implementations of the above classes that are saved in the D2L library. We strongly recommend that you look at the implementation in detail once you have gained some more familiarity with deep learning modeling.
2. Remove the save_hyperparameters statement in the B class. Can you still print self.a and self.b? Optional: if you have dived into the full implementation of the HyperParameters class, can you explain why?

1. Locate full implementations of the above classes that are saved in the D2L library. We strongly recommend that you look at the implementation in detail once you have gained some more familiarity with deep learning modeling.

```
import torch
from torch import nn
from d2l import torch as d2l
from torch.utils.data import DataLoader
from torchvision import datasets, transforms


# 3.2.2. Code Edited

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

    def loss(self, y_hat, y):
        return nn.CrossEntropyLoss()(y_hat, y)

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is not defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not initialized'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / ₩
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / ₩
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / ₩
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=0.1)


# 3.2.3. Code Edited

class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4, batch_size=64):
        self.save_hyperparameters()
        self.transform = transforms.Compose([transforms.ToTensor()])
        self.batch_size = batch_size

    def get_dataloader(self, train):
        dataset = datasets.FashionMNIST(root=self.root, train=train, transform=self.transform, download=True)
        return DataLoader(dataset, batch_size=self.batch_size, shuffle=train, num_workers=self.num_workers)

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

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


# 3.2.4. Code Edited

class Trainer(d2l.HyperParameters):
    """The base class for training models with data."""
    def __init__(self, max_epochs=10, 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):
        self.model.train()
        for batch in self.train_dataloader:
            self.train_batch_idx += 1
            self.optim.zero_grad()
            l = self.model.training_step(batch)
            l.backward()
            self.optim.step()
        if self.val_dataloader:
            self.model.eval()
            for batch in self.val_dataloader:
                self.model.validation_step(batch)


# A random NN model created just to test out D2L libraries and classes

class SimpleNN(Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )


data = DataModule(batch_size=64)
model = SimpleNN()
trainer = Trainer(max_epochs=5)


# Training & Validating a FashionMNIST dataset by using classes: Module, DataModule, Trainer

trainer.fit(model, data)
```



2. Remove the save_hyperparameters statement in the B class. Can you still print self.a and self.b? Optional: if you have dived into the full implementation of the HyperParameters class, can you explain why?

```
# Error happens when save_hyperparameters statement from B class is gone

class B(d2l.HyperParameters):
    def __init__(self, a, b, 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)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-31-1d7a26e5db1d> in <cell line: 7>()
      5         print('There is no self.c =', not hasattr(self, 'c'))
      6
----> 7 b = B(a=1, b=2, c=3)

<ipython-input-31-1d7a26e5db1d> in __init__(self, a, b, c)
      2 class B(d2l.HyperParameters):
      3     def __init__(self, a, b, c):
----> 4         print('self.a =', self.a, 'self.b =', self.b)
      5         print('There is no self.c =', not hasattr(self, 'c'))
      6

AttributeError: 'B' object has no attribute 'a'
```

Why AttributeError?

- Since `save_hyperparameters()` is a method from class `d2l.HyperParameters`, it saves input a, b, c automatically as self.a, self.b, self.c (saving the parameters to object B)
- However, without `save_hyperparameters()`, a,b,c are nothing but just a local variable of method `__init__` and is not saved for object B.

3.4. Discussion

We Implemented a fully functional neural network model and training loop by:

1. Defining the Model (Simple linear regression model)
2. Defining the Loss Function
3. Defining the Optimization Algorithm **(SGD)**
4. And training!

- In this process, we built a data loader, a model, a loss function, an optimization procedure, and a visualization and monitoring tool.
- We did this by composing a Python object that contains all relevant components for training a model.
- While this is not yet a professional-grade implementation it is perfectly functional and code like this could already help you to solve small problems quickly.

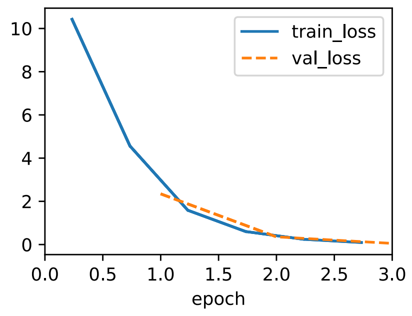
3.4. Exercises & My Own Experiments

1. What would happen if we were to initialize the weights to zero. Would the algorithm still work? What if we initialized the parameters with variance 1000 rather than 0.01?
2. Assume that you are Georg Simon Ohm trying to come up with a model for resistance that relates voltage and current. Can you use automatic differentiation to learn the parameters of your model?
3. Can you use Planck's Law to determine the temperature of an object using spectral energy density? For reference, the spectral density B of radiation emanating from a black body is $B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp \frac{hc}{\lambda kT} - 1\right)^{-1}$. Here λ is the wavelength, T is the temperature, c is the speed of light, h is Planck's constant, and k is the Boltzmann constant. You measure the energy for different wavelengths λ and you now need to fit the spectral density curve to Planck's law.

1-a. What would happen if we were to initialize the weights to zero. Would the algorithm still work?

```
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)


class LinearRegressionZeroInit(LinearRegressionScratch):
    """Linear regression with weights initialized to zero."""
    def __init__(self, num_inputs, lr):
        super().__init__(num_inputs, lr)
        self.w = torch.zeros((num_inputs, 1), requires_grad=True)


model_zero_init = LinearRegressionZeroInit(2, lr=0.03)
trainer_zero_init = d2l.Trainer(max_epochs=3)
```

```
print("Training with weights initialized to zero:")
trainer_zero_init.fit(model_zero_init, data)
```



```
with torch.no_grad():
    print(f'error in estimating w (zero init): {data.w - model_zero_init.w.reshape(data.w.shape)}')
    print(f'error in estimating b (zero init): {data.b - model_zero_init.b}')
```

```
error in estimating w (zero init): tensor([ 0.1567, -0.1764])
error in estimating b (zero init): tensor([0.2383])
```

Why initializing weight to 0 is a bad idea:

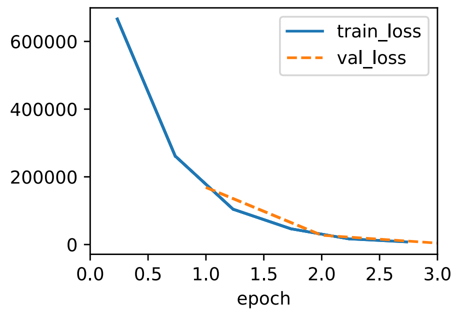
- If we initialize the weights to zero, the linear regression model would still converge and work, but the learning process is **slow at the beginning** of the training process, and less efficient.

1-b. What if we initialized the parameters with variance 1000 rather than 0.01?

```
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
```

```
class LinearRegressionLargeVariance(LinearRegressionScratch):
    """Linear regression with weights initialized with large variance."""
    def __init__(self, num_inputs, lr):
        super().__init__(num_inputs, lr)
        self.w = torch.normal(0, 1000, (num_inputs, 1), requires_grad=True)
```

```
model_large_variance = LinearRegressionLargeVariance(2, lr=0.03)
trainer_large_variance = d2l.Trainer(max_epochs=3)
print("WnTraining with weights initialized with large variance:")
trainer_large_variance.fit(model_large_variance, data)
```



```
with torch.no_grad():
    print(f'error in estimating w (large variance): {data.w - model_large_variance.w.reshape(data.w.shape)}')
    print(f'error in estimating b (large variance): {data.b - model_large_variance.b}')
```

```
error in estimating w (large variance): tensor([-75.5174,  53.4922])
error in estimating b (large variance): tensor([-3.7389])
```

2 main reasons why parameters with large variance is a bad idea:

- **Gradient Explosion:** During the first few iterations, the loss might be very large, leading to large gradients, which in turn may cause the weights to update too drastically, causing the model to diverge.
- **Instability in Training:** The optimization process might become unstable, as the large initial weights could cause the model's predictions to be far off, increasing the chance of overflows or numerical instability during training.

---

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

```
true_R = 5.0
I = torch.tensor([0.5, 1.0, 1.5, 2.0, 2.5])
V_true = I * true_R

R = torch.tensor(1.0, requires_grad=True)

learning_rate = 0.01
num_epochs = 100

for epoch in range(num_epochs):
    V_pred = I * R
    loss = torch.mean((V_pred - V_true) ** 2)
    loss.backward()

    with torch.no_grad():
        R -= learning_rate * R.grad
        R.grad.zero_()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}, Estimated R: {R.item():.4f}')

print(f'WnEstimated resistance: {R.item():.4f} ohms')
```

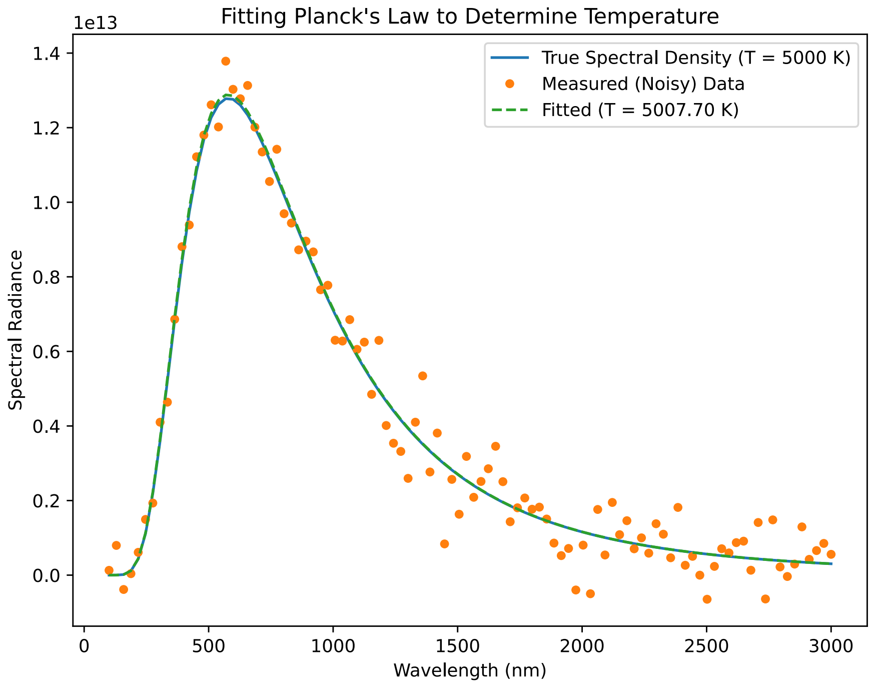
```
Epoch [10/100], Loss: 15.8937, Estimated R: 2.7282
Epoch [20/100], Loss: 5.1270, Estimated R: 3.7097
Epoch [30/100], Loss: 1.6539, Estimated R: 4.2672
Epoch [40/100], Loss: 0.5335, Estimated R: 4.5838
Epoch [50/100], Loss: 0.1721, Estimated R: 4.7636
Epoch [60/100], Loss: 0.0555, Estimated R: 4.8657
Epoch [70/100], Loss: 0.0179, Estimated R: 4.9237
Epoch [80/100], Loss: 0.0058, Estimated R: 4.9567
Epoch [90/100], Loss: 0.0019, Estimated R: 4.9754
Epoch [100/100], Loss: 0.0006, Estimated R: 4.9860

Estimated resistance: 4.9860 ohms
```

---

3. Can you use Planck's Law to determine the temperature of an object using spectral energy density? For reference, the spectral density B of radiation emanating from a black body is $B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp\frac{hc}{\lambda kT} - 1\right)^{-1}$. Here λ is the wavelength, T is the temperature, c is the speed of light, h is Planck's constant, and k is the Boltzmann constant. You measure the energy for different wavelengths λ and you now need to fit the spectral density curve to Planck's law.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

h = 6.626e-34
c = 3e8
k = 1.381e-23

def planck_law(wavelength, T):
    """Spectral radiance based on Planck's law."""
    term1 = 2 * h * c**2 / wavelength**5
    term2 = np.exp(h * c / (wavelength * k * T)) - 1
    return term1 / term2

T_true = 5000
wavelengths = np.linspace(1e-7, 3e-6, 100)
B_measured = planck_law(wavelengths, T_true)

B_noisy = B_measured + np.random.normal(0, 0.05 * B_measured.max(), B_measured.shape)

def fit_temperature(wavelengths, B_measured):
    popt, _ = curve_fit(planck_law, wavelengths, B_measured, p0=[3000])
    return popt[0]

T_estimated = fit_temperature(wavelengths, B_noisy)

print(f"Estimated Temperature: {T_estimated:.2f} K (True Temperature: {T_true} K)")

plt.figure(figsize=(8, 6))
plt.plot(wavelengths * 1e9, B_measured, label=f"True Spectral Density (T = {T_true} K)")
plt.plot(wavelengths * 1e9, B_noisy, 'o', label="Measured (Noisy) Data", markersize=4)
plt.plot(wavelengths * 1e9, planck_law(wavelengths, T_estimated), '--', label=f"Fitted (T = {T_estimated:.2f} K)")
plt.xlabel('Wavelength (nm)')
plt.ylabel('Spectral Radiance')
plt.legend()
plt.title("Fitting Planck's Law to Determine Temperature")
plt.show()
```

```
Estimated Temperature: 5007.70 K (True Temperature: 5000 K)
```



By the way, Planck's Law is a **non-linear** relationship between the wavelength λ and the temperature T since it includes exponentials.

COSE474-2024F Deep Learning HW 1

- **Chapter 4**
- 2021170964 박경빈

0.1. Installation

[ ] ↳ *숨겨진 셀 2개*

4.1. Softmax Regression

4.1.1. Classification

If we say each input consists of a 2×2 grayscale image:

- We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4.
- Let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".

One-Hot Encoding

- A 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 y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken", and $(0, 0, 1)$ to "dog":

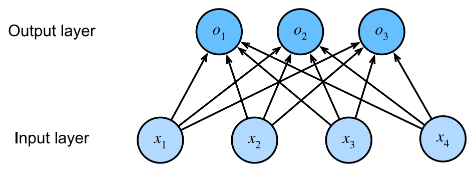
$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

4.1.1.1. Linear Model

Since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts). This yields:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,\\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,\\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned}$$

The corresponding neural network diagram is shown in :numref: `fig_softmaxreg`. Just as in linear regression, we use a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on every input, x_1, x_2, x_3, and x_4, the output layer can also be described as a *fully connected layer.*



4.1.1.2. The Softmax

We could try to minimize the difference between $\mathbf{o}$ and the labels $\mathbf{y}$. However, it is nonetheless unsatisfactory in the following ways:

- 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.

As such, we need a mechanism to "squish" the outputs.

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

Moreover, because the softmax operation preserves the ordering among its arguments, we do not need to compute the softmax to determine which class has been assigned the highest probability. Thus,

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j.$$

4.1.1.3. Vectorization

To improve computational efficiency, we vectorize calculations in minibatches of data. Assume that we are given a minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ of n examples with dimensionality (number of inputs) d. Moreover, assume that we have q categories in the output. Then the weights satisfy $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned}$$

- This accelerates the dominant operation into a matrix--matrix product $\mathbf{X}\mathbf{W}$.
- Moreover, since each row in $\mathbf{X}$ represents a data example, the softmax operation itself can be computed *rowwise*: for each row of $\mathbf{O}$, exponentiate all entries and then normalize them by the sum.

4.1.2. Loss Function

Now that we have a mapping from features $\mathbf{x}$ to probabilities $\hat{\mathbf{y}}$, we need a way to **optimize the accuracy** of this mapping. We will rely on maximum likelihood estimation.

4.1.2.1. Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as the (estimated) conditional probabilities of each class, given any input $\mathbf{x}$, such as $\hat{y}_1$ = $P(y = \mathrm{cat} \mid \mathbf{x})$. In the following we assume that for a dataset with features $\mathbf{X}$ the labels $\mathbf{Y}$ are represented using a one-hot encoding label vector. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}).$$

- We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution $P(\mathbf{y} \mid \mathbf{x}^{(i)})$.

- We take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

where for any pair of label $\mathbf{y}$ and model prediction $\hat{\mathbf{y}}$ over q classes, the loss function l is

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

4.1.2.2. Softmax and Cross-Entropy Loss

Using the definition of the softmax and cross-entropy we obtain

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

Consider the derivative with respect to any logit o_j. We get

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

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. This places a limit on our ability to compress data. For a distribution P its *entropy*, $H[P]$, is defined as:

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

In order to encode data drawn randomly from the distribution P, we need at least $H[P]$ "nats" to encode it

4.1.3.2. Surprisal

If we cannot perfectly predict every event, then we might sometimes be surprised. Our surprise is greater when an event is assigned lower probability. Claude Shannon settled on $\log \frac{1}{P(j)} = -\log P(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $P(j)$. The **entropy 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

So if entropy is the level of surprise experienced by someone who knows the true probability, then you might be wondering, what is cross-entropy? 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. This is given by $H(P, Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$. The lowest possible cross-entropy is achieved when $P = Q$. In this case, the cross-entropy from P to Q is $H(P, P) = H(P)$.

Cross-entropy classification objective:

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

We will focus our discussion in the coming sections on the qualitatively similar, but much smaller **Fashion-MNIST dataset** (Xiao et al., 2017) which was released in 2017. It contains images of 10 categories of clothing at 28 x 28 pixels resolution.

```
%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

```
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)
```

```
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 [00:01<00:00, 15380673.39it/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, 311218.36it/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 [00:00<00:00, 5480919.57it/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, 5190451.20it/s]Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw


(60000, 10000)
```

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

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

```
@d2l.add_to_class(FashionMNIST)
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]
```

4.2.2. Reading a Minibatch

```
@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,
                                       num_workers=self.num_workers)
```

```
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
```

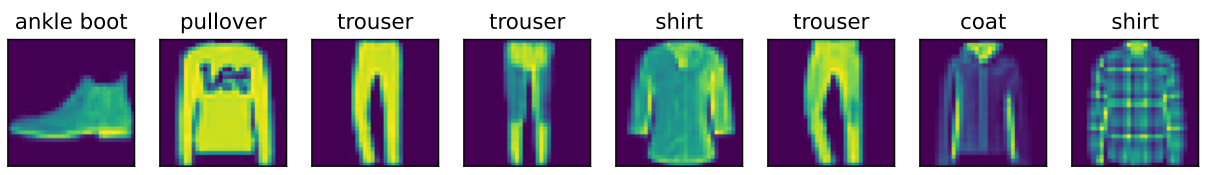
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'13.03 sec'
```

4.2.3. Visualization

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

```
@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)
```



4.3. The Base Classification Model

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

4.3.1. The Classifier Class

```
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)
```

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

4.3.2. Accuracy

```
@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

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

4.4.1. The Softmax

```
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.]]))
```

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

```
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([[0.1382, 0.2614, 0.1485, 0.3026, 0.1493],
         [0.1702, 0.3190, 0.1613, 0.1455, 0.2039]]),
 tensor([1.0000, 1.0000]))
```

4.4.2. The Model

```
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

```
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])
```

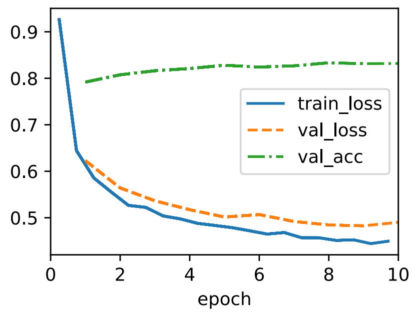
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

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

```
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)
```

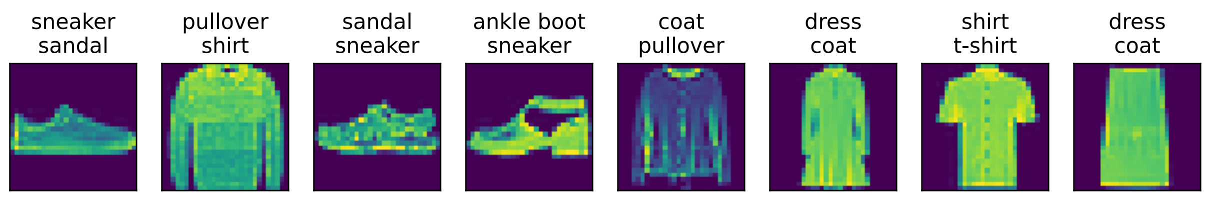


4.4.5. Prediction

```
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+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



Discussions & Exercises

4.1. Discussion

Classification: The task of assigning inputs to discrete categories such as predicting whether an email is spam or a user will subscribe.

One-Hot Encoding: A common method for representing categorical labels where each category is represented by a vector with a single 1 and the rest 0s.

Linear Model for Classification: A model with multiple outputs corresponding to different categories. Each output depends on the input features through an affine transformation. The softmax regression model transforms these outputs into probabilities.

Softmax Function: A normalization technique that transforms raw model outputs (logits) into probabilities by exponentiating them and dividing by their sum. This ensures that the outputs sum to 1 and lie between 0 and 1.

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

Loss Function (Cross-Entropy): The cross-entropy loss is used to compare the predicted probabilities with the true labels, providing a measure of how well the model's predicted distribution matches the actual one.

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

Vectorization: The softmax and cross-entropy loss are computed efficiently by processing data in minibatches, making deep learning models computationally feasible.

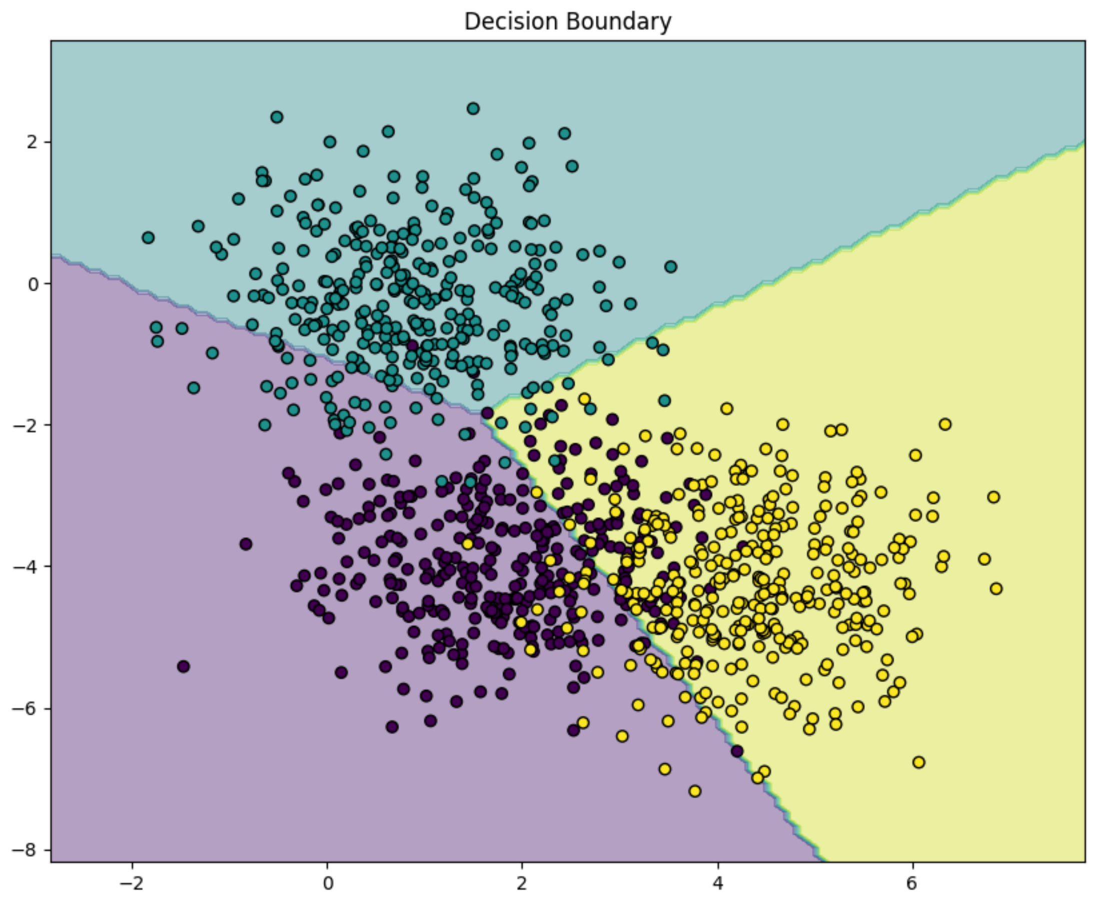
Information Theory Concepts: The text introduces entropy as a measure of uncertainty in a distribution and surprisal as a measure of how "surprised" we are by an event. Cross-entropy combines these ideas to measure how well predicted probabilities match the actual distribution.

4.1 Exercises & My Own Experiment

My Experiment: **Softmax Regression model** to classify data points into *three* clusters.

```
import numpy as np
import matplotlib.pyplot as plt

def generate_data(n_samples, n_features, n_clusters):
    X, y = [], []
    for i in range(n_clusters):
        X.append(np.random.randn(n_samples // n_clusters, n_features) + np.random.randn(n_features) * 3)
        y.append(np.full(n_samples // n_clusters, i))
    return np.vstack(X), np.concatenate(y)

X, y = generate_data(n_samples=1000, n_features=2, n_clusters=3)
y_one_hot = np.eye(3)[y]

def softmax(X):
    exp_X = np.exp(X - np.max(X, axis=1, keepdims=True))
    return exp_X / np.sum(exp_X, axis=1, keepdims=True)

class SoftmaxRegression:
    def __init__(self, input_dim, num_classes):
        self.W = np.random.randn(input_dim, num_classes) * 0.01
        self.b = np.zeros((1, num_classes))

    def forward(self, X):
        return softmax(np.dot(X, self.W) + self.b)

    def train(self, X, y, epochs, lr):
        for epoch in range(epochs):
            y_pred = self.forward(X)
            self.W -= lr * np.dot(X.T, (y_pred - y)) / X.shape[0]
            self.b -= lr * np.mean(y_pred - y, axis=0, keepdims=True)
            if epoch % 10 == 0:
                loss = -np.mean(np.sum(y * np.log(y_pred + 1e-15), axis=1))
                print(f"Epoch {epoch}/{epochs} - Loss: {loss:.4f}")

model = SoftmaxRegression(2, 3)
model.train(X, y_one_hot, epochs=100, lr=0.1)

def plot_results(X, y, model):
    plt.figure(figsize=(10, 8))
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))

    Z = model.forward(np.c_[xx.ravel(), yy.ravel()])
    Z = np.argmax(Z, axis=1).reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.4)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='black')
    plt.title('Decision Boundary')
    plt.show()

plot_results(X, y, model)

accuracy = np.mean(np.argmax(model.forward(X), axis=1) == y)
print(f"Model Accuracy: {accuracy:.2%}")
```

```
Epoch 0/100 - Loss: 1.0687
Epoch 10/100 - Loss: 0.7773
Epoch 20/100 - Loss: 0.6863
Epoch 30/100 - Loss: 0.6273
Epoch 40/100 - Loss: 0.5847
Epoch 50/100 - Loss: 0.5521
Epoch 60/100 - Loss: 0.5260
Epoch 70/100 - Loss: 0.5046
Epoch 80/100 - Loss: 0.4866
Epoch 90/100 - Loss: 0.4712
```



```
Model Accuracy: 83.18%
```

4.2. Discussion

Fashion-MNIST: A more challenging dataset than MNIST that includes 60,000 training and 10,000 test grayscale images of 10 clothing categories.

Fashion-MNIST offers a more challenging task than MNIST while maintaining the same size and simplicity, making it a better benchmark for modern models. Discussing why benchmarking datasets evolve over time is important.

1. **Dataset Benchmarking and Overfitting on Popular Datasets** With the widespread use of datasets like MNIST, researchers might over-optimize for specific datasets, leading to models that perform well only on them. It's worth discussing techniques or practices to avoid overfitting to benchmarks.
2. **Dataset Visualization** The importance of visualizing data, both during exploration and throughout the development cycle, was highlighted. Discuss why human intuition is essential in identifying irregularities in datasets and how visualization helps.
3. **Importance of Diverse and Complex Datasets (Beyond Fashion-MNIST)** While Fashion-MNIST adds more complexity than MNIST, real-world datasets are often far more complex and noisy, requiring models that generalize better. Explore other datasets and strategies for simulating real-world complexity in machine learning tasks.

4.2 Exercises & My Own Experiment

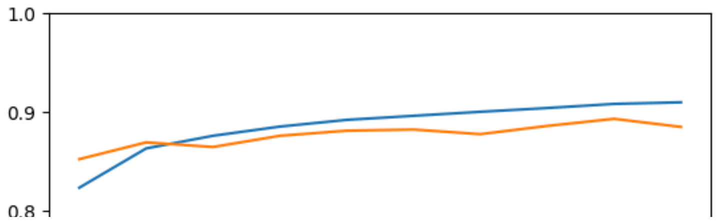
My Experiment: *A simple neural network* for classifying fashion items using the **Fashion MNIST dataset**.

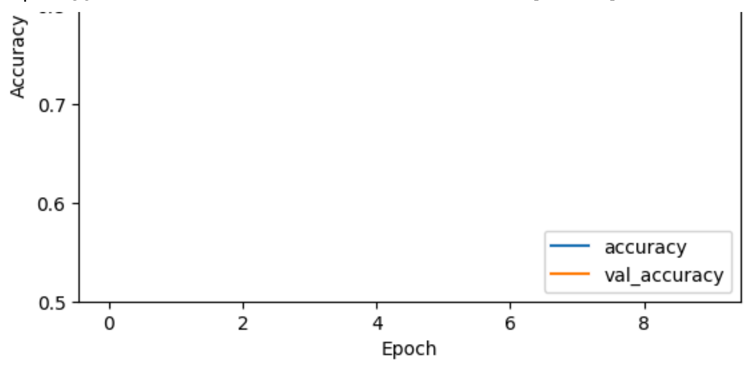
- It loads the data, trains a model, evaluates its performance, and visualizes the results.

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
plt.show()

model = keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=10, validation_split=0.1)

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'₩n테스트 정확도: {test_acc}')

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')
plt.show()

predictions = model.predict(x_test)

def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array[i], true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions, y_test, x_test)
plt.show()
```

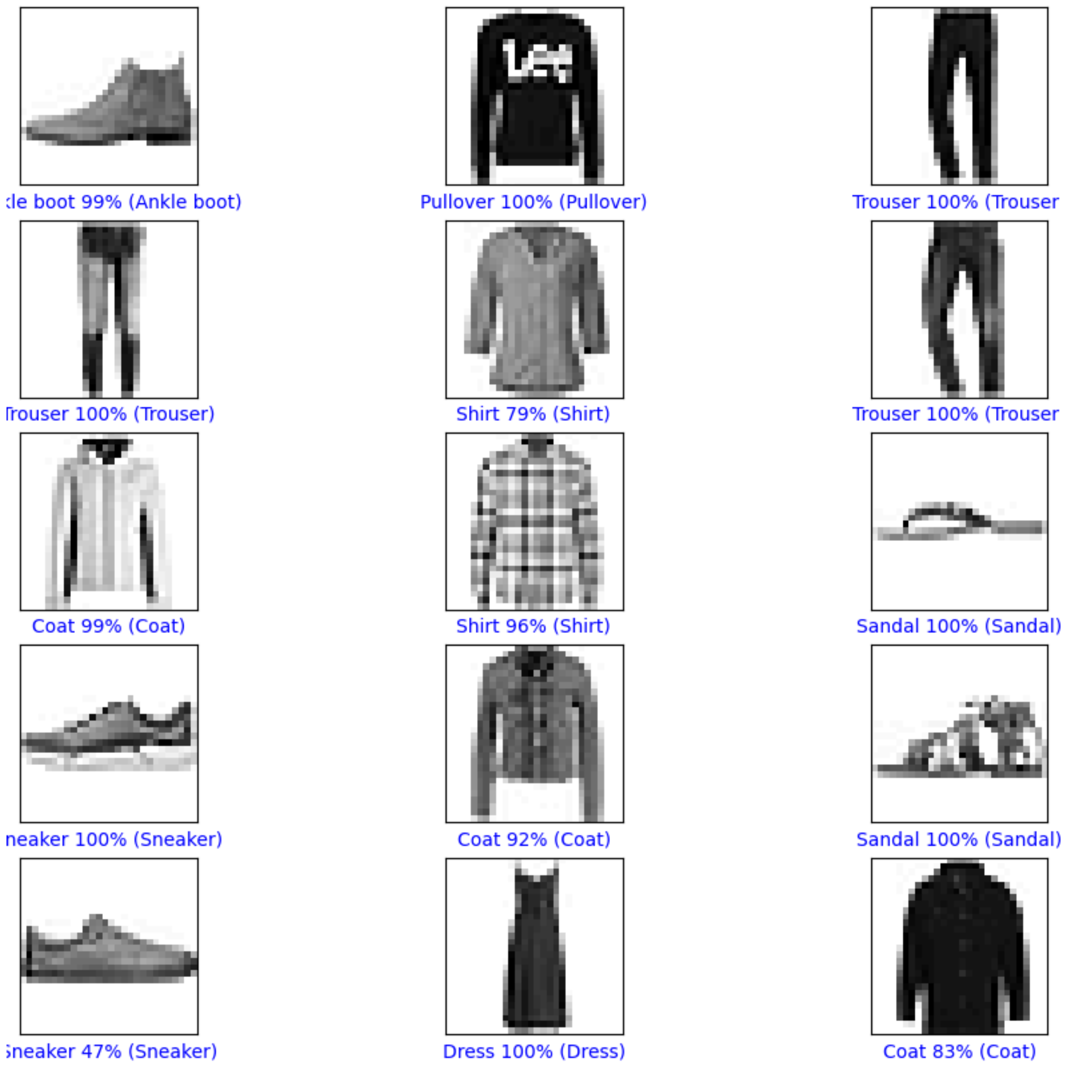
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument
  super().__init__(**kwargs)
Epoch 1/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7808 - loss: 0.6272 - val_accuracy: 0.8522 - val_loss: 0.4186
Epoch 2/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8602 - loss: 0.3949 - val_accuracy: 0.8692 - val_loss: 0.3616
Epoch 3/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8745 - loss: 0.3412 - val_accuracy: 0.8645 - val_loss: 0.3807
Epoch 4/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8843 - loss: 0.3181 - val_accuracy: 0.8758 - val_loss: 0.3331
Epoch 5/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8921 - loss: 0.2984 - val_accuracy: 0.8810 - val_loss: 0.3303
Epoch 6/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8970 - loss: 0.2790 - val_accuracy: 0.8822 - val_loss: 0.3223
Epoch 7/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9013 - loss: 0.2670 - val_accuracy: 0.8775 - val_loss: 0.3489
Epoch 8/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 8s 3ms/step - accuracy: 0.9038 - loss: 0.2593 - val_accuracy: 0.8858 - val_loss: 0.3270
Epoch 9/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9064 - loss: 0.2509 - val_accuracy: 0.8930 - val_loss: 0.3123
Epoch 10/10
1688/1688 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9128 - loss: 0.2341 - val_accuracy: 0.8848 - val_loss: 0.3391
313/313 - 0s - 1ms/step - accuracy: 0.8798 - loss: 0.3588

테스트 정확도: 0.879800021648407
```





```
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
```



The model seems to be performing well in classifying various fashion items. Many predictions show 100% confidence, such as for trousers, pullover, sandal, and dress. However...

- Few Misclassifications: There are some misclassifications. Some notable examples are a shirt being classified with 79% confidence, and a sneaker with 47%.

4.3. Discussion

Classification is a sufficiently common problem that it warrants its own convenience functions.

`Classifier` **Class:** A base class for classification models, inheriting from d2l.Module. Contains a validation_step method, which calculates loss and accuracy on validation data, and plots the results.

`configure_optimizers` **Method:** A method added to d2l.Module for setting up the optimizer (Stochastic Gradient Descent, SGD) with a given learning rate.

Accuracy Calculation: If Y_hat (predictions) is a matrix, the second dimension contains class scores. The predicted class is determined using argmax, which selects the class with the highest score. Predicted classes are compared to ground truth labels Y, and accuracy is computed as the proportion of correct predictions.

4.3 Exercises & My Own Experiment

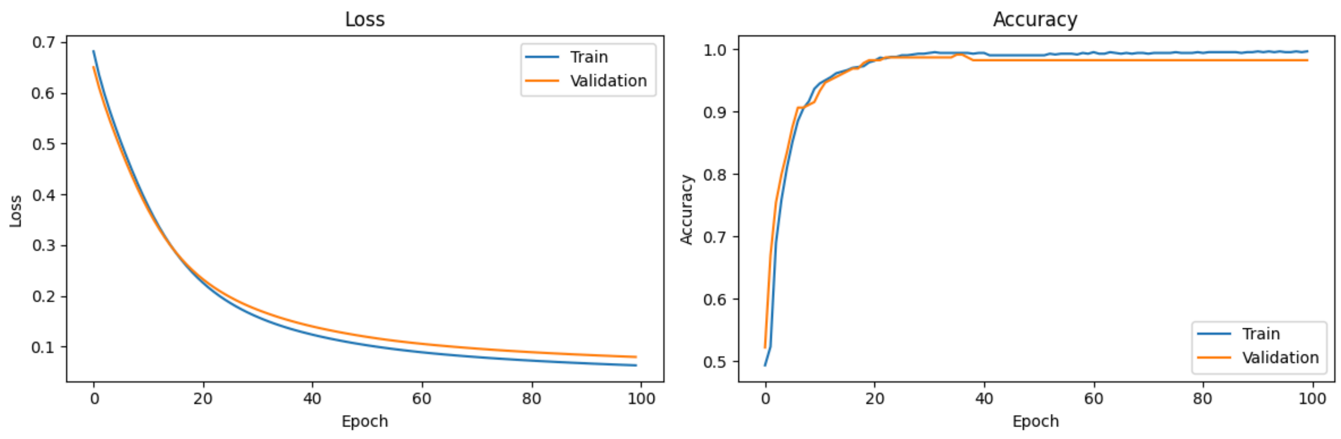
My Experiment: A simple neural network classifier using the Classifier Classes from chapter 4.3:

```
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)


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


@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
```

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

class d2l:
    class Module(nn.Module):
        def __init__(self):
            super().__init__()
            self.lr = 0.01

        def plot(self, name, value, train):
            pass

    @staticmethod
    def add_to_class(Class):
        def wrapper(obj):
            setattr(Class, obj.__name__, obj)
        return wrapper

################# Code from 4.3 #################
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

```
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)

    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
        )
        self.loss = nn.CrossEntropyLoss()

    def forward(self, x):
        return self.model(x)

@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
##################################################

def generate_data(n_samples=1000):
    X = torch.randn(n_samples, 2)
    y = (X[:, 0] + X[:, 1] > 0).long()
    return X, y

def train(model, train_loader, val_loader, epochs=100):
    optimizer = model.configure_optimizers()
    train_losses, train_accs = [], []
    val_losses, val_accs = [], []

    for epoch in range(epochs):
        model.train()
        train_loss, train_acc = 0, 0
        for batch in train_loader:
            optimizer.zero_grad()
            X, y = batch
            y_hat = model(X)
            loss = model.loss(y_hat, y)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()
            train_acc += model.accuracy(y_hat, y).item()

        model.eval()
        val_loss, val_acc = 0, 0
        with torch.no_grad():
            for batch in val_loader:
                model.validation_step(batch)
                X, y = batch
                y_hat = model(X)
                val_loss += model.loss(y_hat, y).item()
                val_acc += model.accuracy(y_hat, y).item()

        train_loss /= len(train_loader)
        train_acc /= len(train_loader)
        val_loss /= len(val_loader)
        val_acc /= len(val_loader)

        train_losses.append(train_loss)
        train_accs.append(train_acc)
        val_losses.append(val_loss)
        val_accs.append(val_acc)

        if (epoch + 1) % 10 == 0 or epoch == 0:
            print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, '
                  f'Train Acc: {train_acc:.4f}, Val Loss: {val_loss:.4f}, '
                  f'Val Acc: {val_acc:.4f}')

    return train_losses, train_accs, val_losses, val_accs

def plot_results(train_losses, train_accs, val_losses, val_accs):
    plt.figure(figsize=(12, 4))

    plt.subplot(121)
    plt.plot(train_losses, label='Train')
    plt.plot(val_losses, label='Validation')
    plt.title('Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(122)
    plt.plot(train_accs, label='Train')
    plt.plot(val_accs, label='Validation')
    plt.title('Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    X, y = generate_data()
    train_size = int(0.8 * len(X))
    X_train, X_val = X[:train_size], X[train_size:]
    y_train, y_val = y[:train_size], y[train_size:]

    train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)
    val_loader = DataLoader(TensorDataset(X_val, y_val), batch_size=32)

    model = Classifier(input_dim=2, hidden_dim=10, output_dim=2)

    train_losses, train_accs, val_losses, val_accs = train(model, train_loader, val_loader)
    plot_results(train_losses, train_accs, val_losses, val_accs)
```

```
Epoch 1/100, Train Loss: 0.6813, Train Acc: 0.4938, Val Loss: 0.6500, Val Acc: 0.5223
Epoch 10/100, Train Loss: 0.3987, Train Acc: 0.9363, Val Loss: 0.3899, Val Acc: 0.9152
Epoch 20/100, Train Loss: 0.2358, Train Acc: 0.9788, Val Loss: 0.2415, Val Acc: 0.9821
Epoch 30/100, Train Loss: 0.1631, Train Acc: 0.9925, Val Loss: 0.1769, Val Acc: 0.9866
Epoch 40/100, Train Loss: 0.1264, Train Acc: 0.9938, Val Loss: 0.1424, Val Acc: 0.9821
Epoch 50/100, Train Loss: 0.1045, Train Acc: 0.9900, Val Loss: 0.1210, Val Acc: 0.9821
Epoch 60/100, Train Loss: 0.0902, Train Acc: 0.9925, Val Loss: 0.1068, Val Acc: 0.9821
Epoch 70/100, Train Loss: 0.0805, Train Acc: 0.9938, Val Loss: 0.0971, Val Acc: 0.9821
Epoch 80/100, Train Loss: 0.0733, Train Acc: 0.9950, Val Loss: 0.0899, Val Acc: 0.9821
Epoch 90/100, Train Loss: 0.0678, Train Acc: 0.9950, Val Loss: 0.0844, Val Acc: 0.9821
Epoch 100/100, Train Loss: 0.0634, Train Acc: 0.9962, Val Loss: 0.0799, Val Acc: 0.9821
```



4.4. Discussion

Softmax Function: The softmax function transforms raw scores into probabilities by exponentiating each input and normalizing by the sum of exponentiated values.

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

Model Definition: The softmax regression model is defined by a weight matrix W and a bias vector b. The model uses these parameters to output a probability distribution over classes for each input example.

Class definition for SoftmaxRegressionScratch:

```
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)
```

Cross-Entropy Loss: This is the loss function commonly used for classification, which computes the negative log-likelihood of the true class's predicted probability.

Cross-entropy implementation:

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

Training the Model: The training loop is reused from linear regression, and hyperparameters like the number of epochs, batch size, and learning rate are adjustable. The Fashion-MNIST dataset is used to train and validate the model.

The training process involves feeding the model through 10 epochs, using a mini-batch of 256 examples at each step.

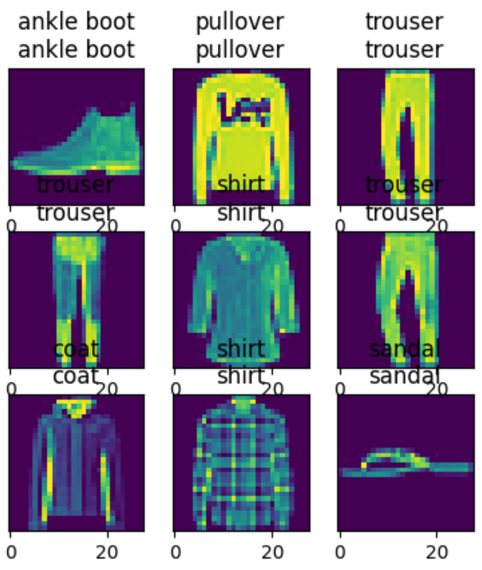
Prediction and Visualization: After training, the model is used to classify images, and incorrect predictions are visualized. This helps assess where the model fails and understand its weaknesses.

4.4 Exercises & My Own Experiment

My Experiment: **A softmax regression model** on the **FashionMNIST dataset** using PyTorch.

```
# Trial and Error 1:

import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

transform = transforms.Compose([transforms.ToTensor()])
train_dataset = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True, transform=transform)

batch_size = 256
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition

class SoftmaxRegression(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super().__init__()
        self.W = torch.nn.Parameter(torch.normal(0, 0.01, size=(num_inputs, num_outputs)))
        self.b = torch.nn.Parameter(torch.zeros(num_outputs))

    def forward(self, X):
        return softmax(torch.matmul(X.reshape((-1, self.W.shape[0])), self.W) + self.b)

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

def accuracy(y_hat, y):
    return (y_hat.argmax(axis=1) == y).float().mean()

def train_epoch(net, train_iter, loss, optimizer):
    net.train()
    for X, y in train_iter:
        optimizer.zero_grad()
        y_hat = net(X)
        l = loss(y_hat, y)
        l.backward()
        optimizer.step()

def evaluate_accuracy(net, data_iter):
    net.eval()
    metric = Accumulator(2)
    with torch.no_grad():
        for X, y in data_iter:
            metric.add(accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]

class Accumulator:
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

num_inputs = 784
num_outputs = 10
net = SoftmaxRegression(num_inputs, num_outputs)

num_epochs, lr = 10, 0.1
loss = cross_entropy
optimizer = torch.optim.SGD(net.parameters(), lr=lr)

for epoch in range(num_epochs):
    train_epoch(net, train_loader, loss, optimizer)
    train_acc = evaluate_accuracy(net, train_loader)
    test_acc = evaluate_accuracy(net, test_loader)
    print(f'epoch {epoch + 1}, train acc {train_acc:.3f}, test acc {test_acc:.3f}')


def get_fashion_mnist_labels(labels):
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]

def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.squeeze().numpy())
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes

X, y = next(iter(test_loader))
trues = get_fashion_mnist_labels(y)
preds = get_fashion_mnist_labels(net(X).argmax(axis=1))
titles = [true + '\n' + pred for true, pred in zip(trues, preds)]

show_images(X[0:9], 3, 3, titles[0:9])
plt.show()
```

```
epoch 1, train acc 0.003, test acc 0.003
epoch 2, train acc 0.003, test acc 0.003
epoch 3, train acc 0.003, test acc 0.003
epoch 4, train acc 0.003, test acc 0.003
epoch 5, train acc 0.003, test acc 0.003
epoch 6, train acc 0.003, test acc 0.003
epoch 7, train acc 0.003, test acc 0.003
epoch 8, train acc 0.003, test acc 0.003
epoch 9, train acc 0.003, test acc 0.003
epoch 10, train acc 0.003, test acc 0.003
```



Why accuracy is so low (0.003):

- Guess 1. Since FashionMNIST images are 28x28 pixels, and softmax regression is too simple to learn their complex patterns. A CNN should be better for handling these kinds of images?
- Guess 2. The learning rate of lr = 0.1 might be too high?
- Guess 3. Loss Function Issue: Rather than manual cross_entropy function, using PyTorch's torch.nn.CrossEntropyLoss() would be better?

I tried fixing the approach using CNN:

```
# Trial and Error 2:

import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True, transform=transform)

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = CNN()
```

COSE474-2024F Deep Learning HW 1

- **Chapter 5**
- 2021170964 박경빈

0.1. Installation

[ ] ↳ 숨겨진 셀 2개

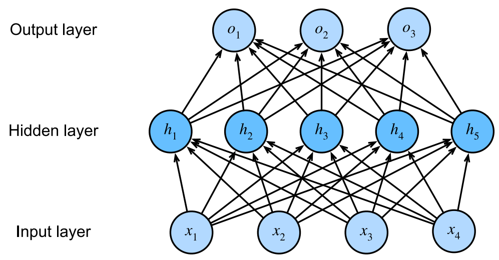
5.1. Multilayer Perceptrons

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

5.1.1. Hidden Layers

5.1.1.1. Limitations of Linear Models

5.1.1.2. Incorporating Hidden Layers



5.1.1.3. From Linear to Nonlinear

As before, we denote by the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ a minibatch of n examples where each example has d inputs (features). For a one-hidden-layer MLP whose hidden layer has h hidden units, we denote by $\mathbf{H} \in \mathbb{R}^{n \times h}$ the outputs of the hidden layer, which are *hidden representations*. Since the hidden and output layers are both fully connected, we have hidden-layer weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ and output-layer weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. This allows us to calculate the outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the one-hidden-layer MLP as follows:

$$\begin{aligned} \mathbf{H} &= \mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned}$$

To see this formally we can just collapse out the hidden layer in the above definition, yielding an equivalent single-layer model with parameters $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X} \mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X} \mathbf{W} + \mathbf{b}.$$

In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear *activation function* σ to be applied to each hidden unit following the affine transformation. For instance, a popular choice is the ReLU (rectified linear unit) activation function :cite:`Nair.Hinton.2010` $\sigma(x) = \mathrm{max}(0, x)$ operating on its arguments elementwise. The outputs of activation functions $\sigma(\cdot)$ are called *activations*. In general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model:

$$\begin{aligned} \mathbf{H} &= \sigma(\mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned}$$

Since each row in $\mathbf{X}$ corresponds to an example in the minibatch, with some abuse of notation, we define the nonlinearity σ to apply to its inputs in a rowwise fashion, i.e., one example at a time.

To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, one atop another, yielding ever more expressive models.

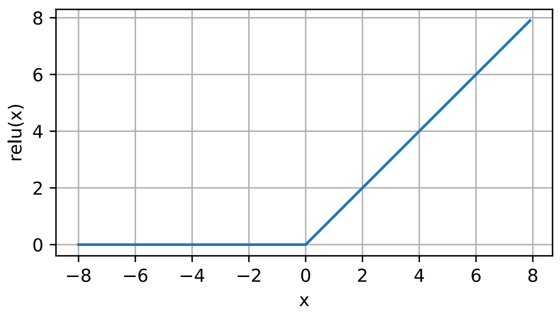
5.1.1.4. Universal Approximators

5.1.2. Activation Functions

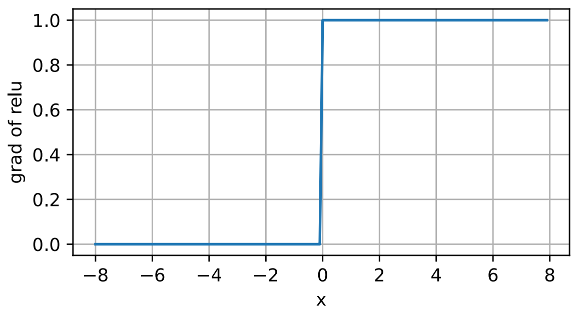
5.1.2.1. ReLU Function

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

```
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))
```

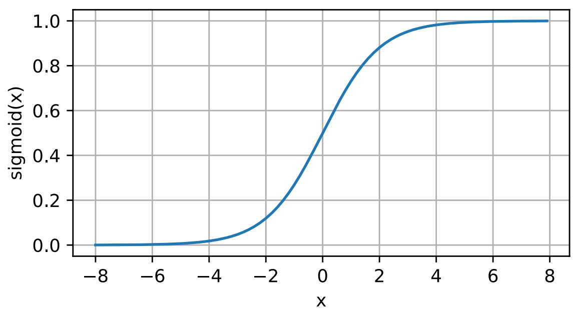


$$\operatorname{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

5.1.2.2. Sigmoid Function

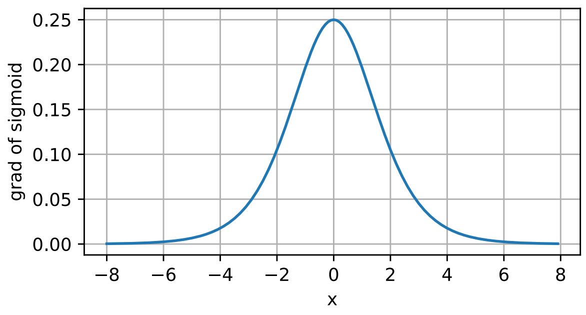
$$\operatorname{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

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



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

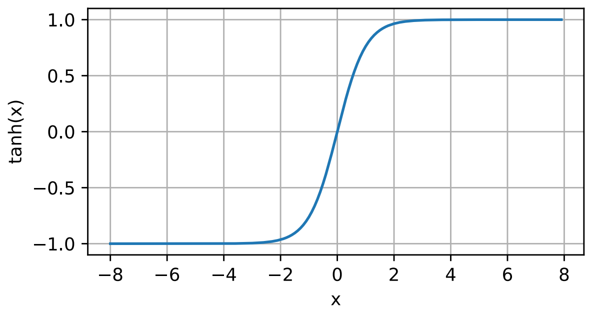
```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



5.1.2.3. Tanh Function

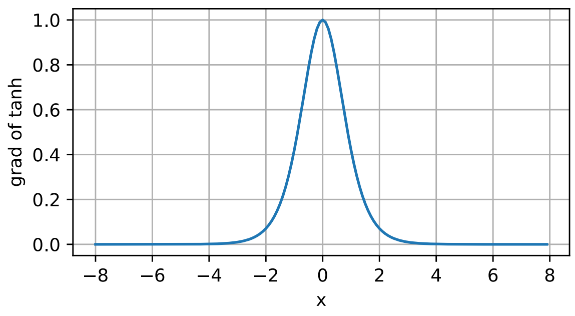
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

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



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

```
# 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

```
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

```
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))
```

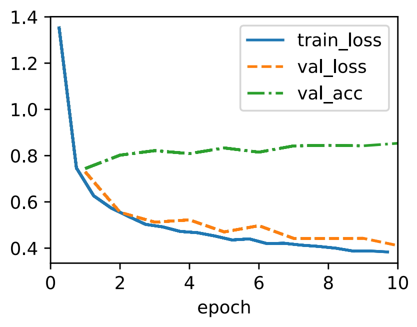
5.2.1.2. Model

```
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

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



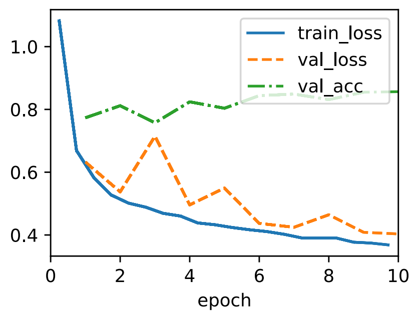
5.2.2. Concise Implementation

5.2.2.1. Model

```
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

```
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 (or *forward pass*) 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. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious but in the eternal words of funk virtuoso James Brown, you must "pay the cost to be the boss".

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h\times d}$ is the weight parameter of the hidden layer. After running the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ through the activation function ϕ we obtain our hidden activation vector of length h:

$$\mathbf{h} = \phi(\mathbf{z}).$$

The hidden layer output $\mathbf{h}$ is also an intermediate variable. Assuming that the parameters of the output layer possess only a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q\times h}$, we can obtain an output layer variable with a vector of length q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

Assuming that the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y).$$

As we will see the definition of ℓ_2 regularization to be introduced later, given the hyperparameter λ, the regularization term is

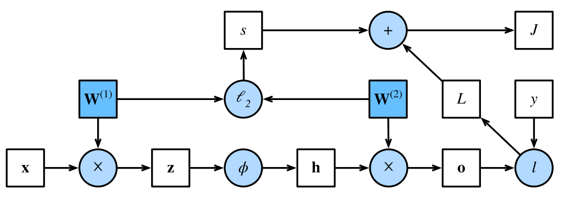
$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_\mathrm{F}^2 + \|\mathbf{W}^{(2)}\|_\mathrm{F}^2\right),$$

where the Frobenius norm of the matrix is simply the ℓ_2 norm applied after flattening the matrix into a vector. Finally, the model's regularized loss on a given data example is:

$$J = L + s.$$

We refer to J as the *objective function* in the following discussion.

5.3.2. Computational Graph of Forward Propagation



5.3.3. Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z with respect to X via

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

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out. For vectors, this is straightforward: it is simply matrix--matrix multiplication. For higher dimensional tensors, we use the appropriate counterpart. The operator prod hides all the notational overhead.

Recall that the parameters of the simple network with one hidden layer, whose computational graph is in :numref: `fig_forward`, are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J=L+s$ with respect to the loss term L and the regularization term s:

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

Next, we compute the gradient of the objective function with respect to variable of the output layer $\mathbf{o}$ according to the chain rule:

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

Next, we calculate the gradients of the regularization term with respect to both parameters:

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

Now we are able to calculate the gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

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

:eqlabel: `eq_backprop-J-h`

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer output $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ is given by

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

Since the activation function ϕ applies elementwise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}$ requires that we use the elementwise multiplication operator, which we denote by $\odot$:

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

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

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

5.3.4. Training Neural Networks

When training neural networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed.

Take the aforementioned simple network as an illustrative example. On the one hand, computing the regularization term :eqref: `eq_forward-s` during forward propagation depends on the current values of model parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. They are given by the optimization algorithm according to backpropagation in the most recent iteration. On the other hand, the gradient calculation for the parameter :eqref: `eq_backprop-J-h` during backpropagation depends on the current value of the hidden layer output $\mathbf{h}$, which is given by forward propagation.

Therefore when training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to *out-of-memory* errors.

Discussions & Exercises

5.1. Discussion

Limitations of Linear Models: Linear models assume monotonicity and fail in tasks with non-linear relationships (e.g., image classification).

Hidden Layers in MLPs: Hidden layers allow MLPs to capture complex patterns by stacking affine transformations and applying nonlinear activation functions (e.g., ReLU).

Activation Functions:

- `ReLU`: Most common, sets negative inputs to 0. Simple and effective.
- `Sigmoid & Tanh`: Squeeze values but suffer from vanishing gradients. ReLU is often preferred for hidden layers. Newer functions like GELU and Swish offer potential improvements in certain tasks.

Universal Approximation: MLPs can approximate any function with enough hidden units, but deeper models are often more efficient.

Apart from *ReLU, Sigmoid, and Tanh*, several other activation functions exists in order to address the limitations of these traditional choices.

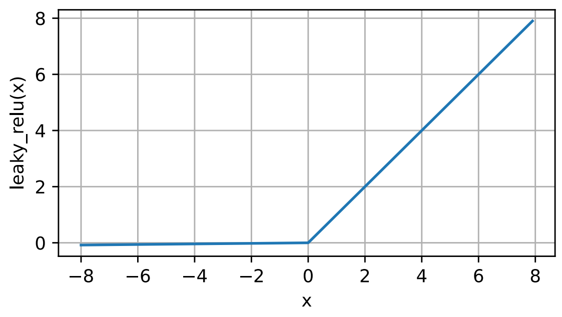
- Might offer alternatives for better performance in specific tasks.

1. Leaky ReLU

- Leaky ReLU is a variant of ReLU, which allows a small, non-zero gradient when the input is negative. This helps avoid "dead neurons" in the network.

$$\text{Leaky ReLU}(x) = \max(\alpha x, x)$$

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.nn.functional.leaky_relu(x, negative_slope=0.01)
d2l.plot(x.detach(), y.detach(), 'x', 'leaky_relu(x)', figsize=(5, 2.5))
```

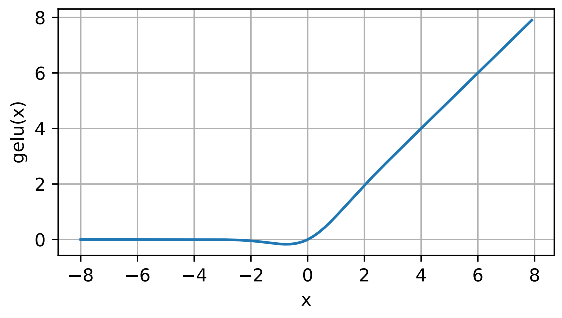


2. GELU (Gaussian Error Linear Unit)

- GELU applies a smoother transition between activation and inactivation compared to ReLU. It's commonly used in transformer models.

$$\text{GELU}(x) = x \cdot \Phi(x)$$

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

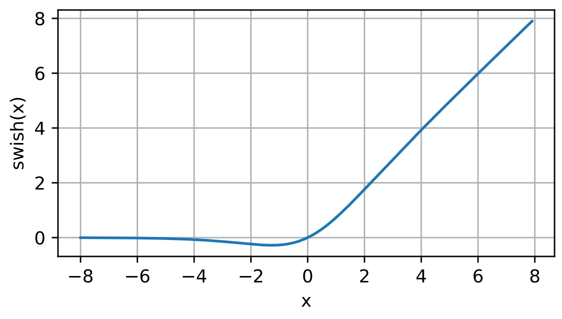


3. Swish

- Swish, like GELU, provides smooth, non-linear transformations. It is defined as:

$$\text{Swish}(x) = x \cdot \sigma(\beta x)$$

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

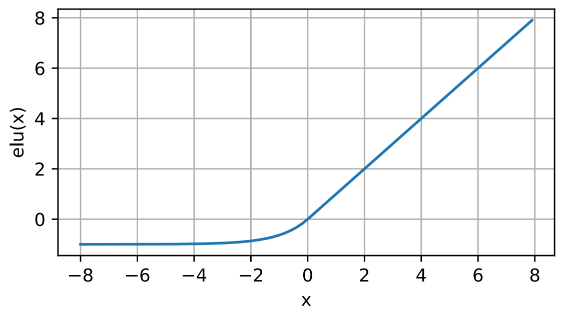


4. ELU (Exponential Linear Unit)

- ELU adds an exponential term for negative values, which helps improve gradient flow and avoid dead neurons.

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha(e^x - 1) & \text{if } x \leq 0. \end{cases}$$

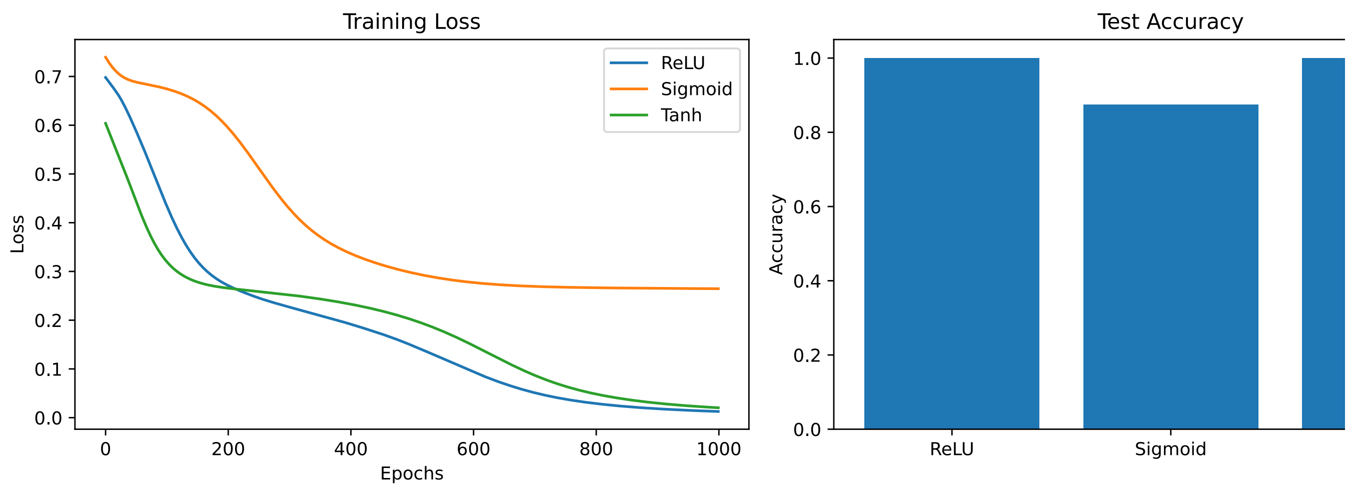
```
y = torch.nn.functional.elu(x, alpha=1.0)
d2l.plot(x.detach(), y.detach(), 'x', 'elu(x)', figsize=(5, 2.5))
```



5.1 Exercises & My Own Experiments

I wonder what happens if I compare ReLU v/s Sigmoid v/s Tanh function?

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim

X, y = make_moons(n_samples=1000, noise=0.1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_test = torch.FloatTensor(X_test)
y_test = torch.LongTensor(y_test)

class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, activation):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.activation = activation

    def forward(self, x):
        x = self.activation(self.fc1(x))
        x = self.activation(self.fc2(x))
        x = self.fc3(x)
        return x

def train_model(model, X_train, y_train, epochs=1000):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters())

    losses = []
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
        losses.append(loss.item())

    return losses

def evaluate_model(model, X_test, y_test):
    model.eval()
    with torch.no_grad():
        outputs = model(X_test)
        _, predicted = torch.max(outputs, 1)
        accuracy = (predicted == y_test).float().mean()
    return accuracy.item()

input_size = 2
hidden_size = 10
output_size = 2
epochs = 1000

activation_functions = {
    'ReLU': nn.ReLU(),
    'Sigmoid': nn.Sigmoid(),
    'Tanh': nn.Tanh()
}

results = {}

for name, activation in activation_functions.items():
    model = MLP(input_size, hidden_size, output_size, activation)
    losses = train_model(model, X_train, y_train, epochs)
    accuracy = evaluate_model(model, X_test, y_test)
    results[name] = {'losses': losses, 'accuracy': accuracy}

plt.figure(figsize=(12, 4))

plt.subplot(121)
for name, data in results.items():
    plt.plot(data['losses'], label=name)
plt.title('Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.subplot(122)
names = list(results.keys())
accuracies = [data['accuracy'] for data in results.values()]
plt.bar(names, accuracies)
plt.title('Test Accuracy')
plt.ylabel('Accuracy')

plt.tight_layout()
plt.show()

for name, data in results.items():
    print(f"{name} - Final Loss: {data['losses'][-1]:.4f}, Test Accuracy: {data['accuracy']:.4f}")
```



```
ReLU - Final Loss: 0.0127, Test Accuracy: 1.0000
Sigmoid - Final Loss: 0.2646, Test Accuracy: 0.8750
Tanh - Final Loss: 0.0203, Test Accuracy: 1.0000
```

ReLU and Tanh converged faster than Sigmoid, with ReLU slightly better in later epochs.

- ReLU's success: Avoids vanishing gradients, allows sparse activations.
- Tanh's strong showing: Symmetry around zero beneficial for this dataset.
- Sigmoid's struggle: Suffers from vanishing gradients and non-zero-centered output.

5.2. Discussion

With experience in designing deep networks, **adding multiple layers becomes easier**, especially since we can *reuse* the training algorithms.

MLP from scratch:

- Initializes two sets of weights and biases for a simple MLP with one hidden layer.
- Uses a custom ReLU activation function.
- The forward pass reshapes the input, applies the ReLU activation to the hidden layer, and calculates the output.
- Training: The model is trained for 10 epochs using the d2l.Trainer and the Fashion-MNIST dataset.

```
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))

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
```

Concise MLP with PyTorch's API:

- Uses PyTorch’s nn.Sequential to build the model, making it simpler and more flexible.
- Includes layers for flattening the input, applying a hidden layer with ReLU, and outputting predictions.
- Training: The same d2l.Trainer is used to fit this model with the Fashion-MNIST data.

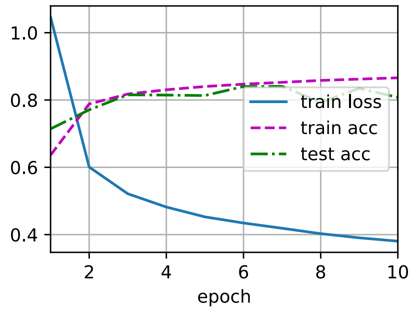
```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))
```

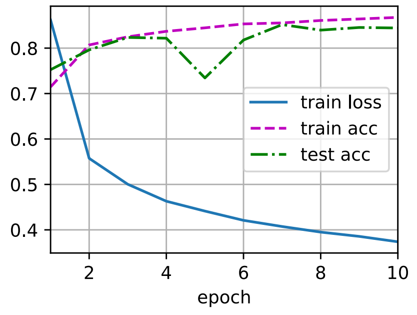
5.2 Exercises & My Own Experiment

MLP from scratch v/s **Concise MLP with PyTorch's API**??

```
import torch
import torch.nn as nn
import torchvision
from torch.utils.data import DataLoader
from d2l import torch as d2l

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))

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

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=self.lr)

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))

    def forward(self, X):
        return self.net(X)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=self.lr)

def load_data_fashion_mnist(batch_size, resize=None):
    trans = [torchvision.transforms.ToTensor()]
    if resize:
        trans.insert(0, torchvision.transforms.Resize(resize))
    trans = torchvision.transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (DataLoader(mnist_train, batch_size, shuffle=True,
                       num_workers=4),
            DataLoader(mnist_test, batch_size, shuffle=False,
                       num_workers=4))

def accuracy(y_hat, y):
    """Compute the number of correct predictions."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())

def train_epoch(model, train_iter, loss, optimizer):
    model.train()
    metric = d2l.Accumulator(3)
    for X, y in train_iter:
        y_hat = model(X)
        l = loss(y_hat, y)
        optimizer.zero_grad()
        l.backward()
        optimizer.step()
        metric.add(float(l) * X.shape[0], accuracy(y_hat, y), X.shape[0])
    return metric[0] / metric[2], metric[1] / metric[2]

def evaluate_accuracy(model, data_iter):
    model.eval()
    metric = d2l.Accumulator(2)
    with torch.no_grad():
        for X, y in data_iter:
            metric.add(accuracy(model(X), y), y.numel())
    return metric[0] / metric[1]

def train(model, train_iter, test_iter, loss, num_epochs):
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    optimizer = model.configure_optimizers()
    for epoch in range(num_epochs):
        train_loss, train_acc = train_epoch(model, train_iter, loss, optimizer)
        test_acc = evaluate_accuracy(model, test_iter)
        animator.add(epoch + 1, (train_loss, train_acc, test_acc))
    return animator

batch_size, num_epochs = 256, 10
num_inputs, num_outputs, num_hiddens = 784, 10, 256
lr = 0.1
train_iter, test_iter = load_data_fashion_mnist(batch_size)
loss = nn.CrossEntropyLoss()
model_scratch = MLPScratch(num_inputs, num_outputs, num_hiddens, lr)
animator_scratch = train(model_scratch, train_iter, test_iter, loss, num_epochs)
model = MLP(num_outputs, num_hiddens, lr)
animator = train(model, train_iter, test_iter, loss, num_epochs)


print(f"MLPScratch - Final test accuracy: {animator_scratch.Y[2][-1]:.4f}")
print(f"MLP - Final test accuracy: {animator.Y[2][-1]:.4f}")
d2l.plt.show()
```

```
MLPScratch - Final test accuracy: 0.8077
MLP - Final test accuracy: 0.8443
```





MLP using d2l is slightly better than MLP from scratch, and its even easier to code!

5.3. Discussion

Forward propagation sequentially calculates and stores intermediate variables within the computational graph defined by the neural network. It proceeds from the input to the output layer.

Backpropagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.

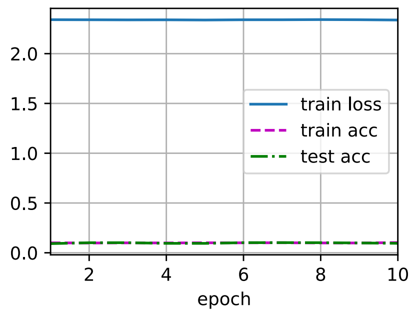
When training deep learning models, forward propagation and backpropagation are interdependent, and training requires significantly more memory than prediction.

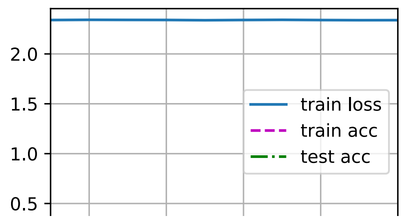
5.3 Exercises & My Own Experiments

What **if forward & backward propagation didn't exist**? I reused code from 5.2 exercises & experiment:

```
import torch
import torch.nn as nn
import torchvision
from torch.utils.data import DataLoader
from d2l import torch as d2l

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))

    def forward(self, X):
        # returns random values rather than forward propagation!!
        return torch.rand(X.shape[0], self.num_outputs)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=self.lr)

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))

    def forward(self, X):
        # returns random values rather than forward propagation!!
        return torch.rand(X.shape[0], self.num_outputs)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=self.lr)

def load_data_fashion_mnist(batch_size, resize=None):
    trans = [torchvision.transforms.ToTensor()]
    if resize:
        trans.insert(0, torchvision.transforms.Resize(resize))
    trans = torchvision.transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (DataLoader(mnist_train, batch_size, shuffle=True,
                       num_workers=4),
            DataLoader(mnist_test, batch_size, shuffle=False,
                       num_workers=4))

def accuracy(y_hat, y):
    """Compute the number of correct predictions."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())

def train_epoch(model, train_iter, loss, optimizer):
    model.train()
    metric = d2l.Accumulator(3)
    for X, y in train_iter:
        y_hat = model(X)
        l = loss(y_hat, y)
        # removed back propagation!!
        metric.add(float(l) * X.shape[0], accuracy(y_hat, y), X.shape[0])
    return metric[0] / metric[2], metric[1] / metric[2]

def evaluate_accuracy(model, data_iter):
    model.eval()
    metric = d2l.Accumulator(2)
    with torch.no_grad():
        for X, y in data_iter:
            metric.add(accuracy(model(X), y), y.numel())
    return metric[0] / metric[1]

def train(model, train_iter, test_iter, loss, num_epochs):
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    optimizer = model.configure_optimizers()
    for epoch in range(num_epochs):
        train_loss, train_acc = train_epoch(model, train_iter, loss, optimizer)
        test_acc = evaluate_accuracy(model, test_iter)
        animator.add(epoch + 1, (train_loss, train_acc, test_acc))
    return animator

batch_size, num_epochs = 256, 10
num_inputs, num_outputs, num_hiddens = 784, 10, 256
lr = 0.1
train_iter, test_iter = load_data_fashion_mnist(batch_size)
loss = nn.CrossEntropyLoss()
model_scratch = MLPScratch(num_inputs, num_outputs, num_hiddens, lr)
animator_scratch = train(model_scratch, train_iter, test_iter, loss, num_epochs)
model = MLP(num_outputs, num_hiddens, lr)
animator = train(model, train_iter, test_iter, loss, num_epochs)

print(f"MLPScratch - Final test accuracy: {animator_scratch.Y[2][-1]:.4f}")
print(f"MLP - Final test accuracy: {animator.Y[2][-1]:.4f}")
d2l.plt.show()
```

```
MLPScratch - Final test accuracy: 0.0951
MLP - Final test accuracy: 0.1015
```





By this experiment, I learned that if front & back propagation doesn't exist, the model basically can't learn anything!
Front & back propagation are CRUCIAL for MLP!