COSE474-2024F: DEEP LEARNING HW1

2. Preliminaries

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

Show hidden output

```
import torch
```

2.1 Data Manipulation

2.1.1. Getting Started

```
x = torch.arange(6, dtype=torch.float32)
x
```

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

```
x.numel()
```

```
6
```

```
x.shape
```

```
torch.Size([6])
```

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

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

```
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([[-0.0518, -1.0996, -0.0491, -0.2285],
        [-0.1701, -0.1580, -1.8895, -0.0241],
        [-0.2280,  0.2207, -1.7376, -0.2960]])
```

```
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 = torch.randn(3, 4)
X[-1], X[1:3]
```

```
(tensor([-1.5078,  1.3323,  1.2738,  0.6004]),
 tensor([[-0.9168,  0.5597, -0.7916, -1.3244],
         [-1.5078,  1.3323,  1.2738,  0.6004]]))
```

```
X[1, 2] = 0
X[:, 3] = 0
X
```

```
tensor([[-1.3903, -0.2266, -0.2376,  0.0000],
        [-0.9168,  0.5597,  0.0000,  0.0000],
        [-1.5078,  1.3323,  1.2738,  0.0000]])
```

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

```
tensor([[12.0000, 12.0000, 12.0000, 12.0000],
        [12.0000, 12.0000, 12.0000, 12.0000],
        [-1.5078,  1.3323,  1.2738,  0.0000]])
```

2.1.3. Operations

```
x = torch.arange(5, dtype=torch.float32)
x
```

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

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

```
tensor([ 1.0000,  2.7183,  7.3891, 20.0855, 54.5981])
```

```
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

```
id(Y)
```

```
135947083788784
```

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

```
False
```

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

```
id(Z): 135947083659952
id(Z): 135947083659952
```

```
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_reverse = B.numpy()
type(A), type(A_reverse)
```

```
(numpy.ndarray, numpy.ndarray)
```

```
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

```
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

```
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

```
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()
A, A + B
```

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

```
A * B
```

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

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

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

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

2.3.6. Reduction

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

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

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

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

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

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

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

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

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

```
tensor(True)
```

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

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

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

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

2.3.7. Non-Reduction Sum

```
A
```

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

```
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_larger = torch.arange(12).reshape(4, 3)
A_larger
```

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

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

```
tensor([[ 0,  1,  2],
        [ 3,  5,  7],
        [ 9, 12, 15],
        [18, 22, 26]])
```

2.3.8. Dot Products

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

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

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

```
tensor(3.)
```

2.3.9. Matrix−Vector Products

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

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

2.3.10. Matrix–Matrix Multiplication

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

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

2.3.11. Norms

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

```
tensor(5.)
```

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

```
tensor(7.)
```

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

```
tensor(6.)
```

2.5. Automatic Differentiation (autograd)

2.5.1. A Simple Function

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

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

```
x.requires_grad_(True)
x.grad
```

```
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_()
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)
a
```

```
tensor(-0.1114, requires_grad=True)
```

```
d = f(a)
d.backward()
```

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

```
tensor(True)
```

2. Discussions & Exercises

2.1 Data Manipulation Summary (Discussion)

Vectors, matrices and multidimensional tensors can be

- created using functions like **arange**, **zeros**, **ones** and **randn**
- concatinated using **cat**
    - torch.cat((X, Y), dim=0) -> concatenate two matrices along rows (axis 0)
    - torch.cat((X, Y), dim=1) -> concatenate two matrices along columns (axis 1)
- reshaped using **reshape** function
- converted between tensor and NumPy types using **numpy** and **from_numpy** functions
- converted to python scalar using **item** function (1D tensor only)

2.1.4. Broadcasting Mechanism (Discussion)

- Even when **shapes differ**, we can still perform **elementwise binary operations** by invoking the **broadcasting mechanism**:

1. expand arrays by copying elements along axes with smaller length so they have the same shape
2. elementwise operation

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

```
a.shape, b.shape, (a + b).shape, a + b
```

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

2.1.5 Dereferencing and Allocating New Memory (Discussion)

- if we write $Y = X + Y$, we dereference the tensor that Y used to point to and instead **point Y at the newly allocated memory**
- In-place updates are crucial for **efficient memory usage** and **consistent parameter updates** by directly modifying the original tensor without allocating new memory.

```
X = torch.arange(3)
Y = torch.arange(3)

# in-place update: YES (2 ways)
before = id(Y)
Y[:] = Y + X
print('Y[:] = Y + X:', id(Y) == before)

before = id(Y)
Y += X
print('Y += X:', id(Y) == before)

# in-place update: NO
before = id(Y)
Y = Y + X
print('Y = Y + X:', id(Y) == before)
```

```
Y[:] = Y + X: True
Y += X: True
Y = Y + X: False
```

2.1.8 Exercises

```
# Ex 1
X = torch.arange(3) + 1
Y = torch.arange(3)


print(f"X: {X}, Y: {Y}")
print("X > Y:", X > Y)
print("X == Y", X == Y)
print("X < Y:", X < Y)
```

```
X: tensor([1, 2, 3]), Y: tensor([0, 1, 2])
X > Y: tensor([True, True, True])
X == Y tensor([False, False, False])
X < Y: tensor([False, False, False])
```

```
# Ex 2
X = torch.arange(16).reshape(4,2,2)
Y = torch.arange(4).reshape(1,2,2)

X, Y, X + Y
# Y copies the values 4 times in the 0 Axis
```

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

         [[ 4,  5],
          [ 6,  7]],

         [[ 8,  9],
          [10, 11]],

         [[12, 13],
          [14, 15]]]),
 tensor([[[0, 1],
          [2, 3]]]),
 tensor([[[ 0,  2],
          [ 4,  6]],

         [[ 4,  6],
          [ 8, 10]],

         [[ 8, 10],
          [12, 14]],

         [[12, 14],
          [16, 18]]]))
```

2.2.2 Data Preparation (Discussion)

Selecting Columns

Selecting columns by:

- integer-location based indexing (iloc)
- *name*

```
data = pd.read_csv(data_file)

data.iloc[:, 1:3], data[["RoofType", "Price"]]
```

```
(  RoofType   Price
 0      NaN  127500
```

```
1       NaN  106000
2     Slate  178100
3       NaN  140000,
   RoofType   Price
0       NaN  127500
1       NaN  106000
2     Slate  178100
3       NaN  140000)
```

Converting categorical variable

pandas.get_dummies:

- variable is converted in as many 0/1 variables as there are different values
- columns are each named after the value, the name of the original variable is prepended to the value

```
inputs = data.iloc[:, 0:2]
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
```

Filling in missing values

- a common heuristic is to replace the NaN entries with the **mean value**

```
inputs
```

	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

Next steps: Generate code with inputs | View recommended plots | New interactive sheet

```
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
```

2.3.5 Hadamard Product (Discussion)

Hadamard Product = the **elementwise** product of two matrices

- denoted $x \odot y$

$$A \odot B = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{pmatrix}$$

2.3.7. Convert row to sum up to 1 (Exercise)

```
test = torch.arange(12).reshape(4, 3).T
# keepdims=True to keep the number of axes unchanged
sum_test = test.sum(axis=1, keepdims=True)
test / sum_test
```

```
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.0455, 0.1818, 0.3182, 0.4545],
        [0.0769, 0.1923, 0.3077, 0.4231]])
```

2.3.9. Matrix–Vector Products (Discussion)

- to perform matrix–vector product we can use:
  - the mv function
  - the @ operator (can execute both matrix–vector and matrix–matrix products)

+ Code + Text

```
A = torch.arange(12, dtype=torch.float32).reshape(4, 3)
x = torch.arange(3, dtype=torch.float32)
A, x
A.shape, x.shape, torch.mv(A, x), A@x
```

```
(torch.Size([4, 3]),
 torch.Size([3]),
 tensor([ 5., 14., 23., 32.]),
 tensor([ 5., 14., 23., 32.]))
```

2.3.11. Norms (Discussion)

- **L1 Norm**:

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

- **L2 Norm**:

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

- **Frobenius Norm**:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}$$

- the **norm** function calculates Frobenius norm for a matrix and L2 norm for a vector (L2 is a specific case of the Frobenius norm for vectors)

2.5. Automatic Differentiation (Discussion)

- as we pass data through each successive function, the framework builds a **computational graph**
- to calculate derivatives, autograd works backwards through this graph applying the chain rule (the algorithm is called **backpropagation**)

- **requires_grad=True** allocates a buffer to store the gradient for the tensor. The buffer is reused in subsequent computations
- **grad.zero_()** to clear the gradient buffer

```
# requires_grad=True
x = torch.arange(4.0, requires_grad=True)
print(x.grad == None)
```

```
True
```

```
fn = 3 * (x ** 3)
fn
```

```
tensor([ 0.,  3., 24., 81.], grad_fn=<MulBackward0>)
```

- invoking backward on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar

```
# fn.backward() -> ERROR
```

- **backward(gradient=torch.ones(len(fn)))** computes the gradient of the tensor fn
- in this example the gradient is $9x^2$ for $x = (0, 1, 2, 3)$

$$grad(x) = (0, 9, 36, 81)$$

```
fn.backward(gradient=torch.ones(len(fn)))
x.grad
```

```
tensor([ 0.,  9., 36., 81.])
```

Autograd detach() Function *Exercises*

```
x = torch.arange(4.0, requires_grad=True)
# grad will be -> 3
y = x * 3
y.backward(gradient=torch.ones(len(y)), retain_graph=True)
y, x, x.grad
```

```
(tensor([0., 3., 6., 9.], grad_fn=<MulBackward0>),
 tensor([0., 1., 2., 3.], requires_grad=True),
 tensor([3., 3., 3., 3.]))
```

```
x.grad.zero_()
# grad will be -> 6x
z = y * x
z.sum().backward()
z, x, x.grad
```

```
(tensor([ 0.,  3., 12., 27.], grad_fn=<MulBackward0>),
 tensor([0., 1., 2., 3.], requires_grad=True),
 tensor([ 0.,  6., 12., 18.]))
```

```
x.grad.zero_()
u = y.detach()
# grad will be -> u = y = [0., 3., 6., 9.]
z = u * x
z.sum().backward()
z, x, x.grad
```

```
(tensor([ 0.,  3., 12., 27.], grad_fn=<MulBackward0>),
 tensor([0., 1., 2., 3.], requires_grad=True),
 tensor([0., 3., 6., 9.]))
```

COSE474-2024F: DEEP LEARNING HW1

3. Linear Neural Networks for Regression

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

Show hidden output

3.1. Linear Regression

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

3.1.2. Vectorization for Speed¶

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

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

```
'0.31065 sec'
```

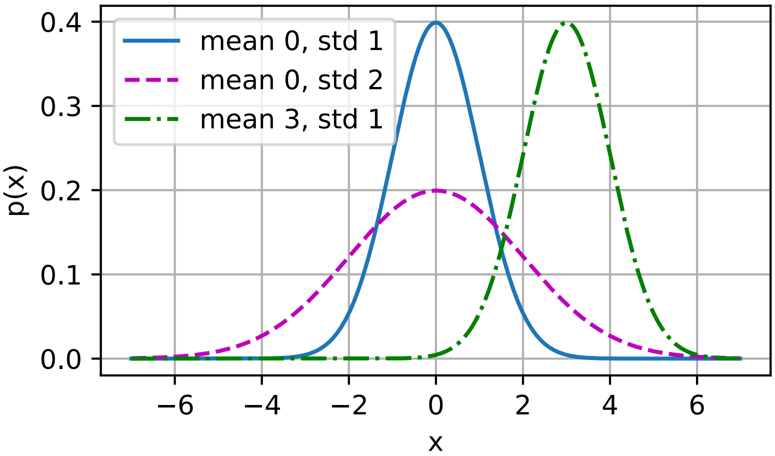
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
'0.00121 sec'
```

3.1.3. The Normal Distribution and Squared Loss

```
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)


x = np.arange(-7, 7, 0.01)

params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



3.2. Object-Oriented Design for Implementation

```
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
    (after the class has been created).
    """

    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
```

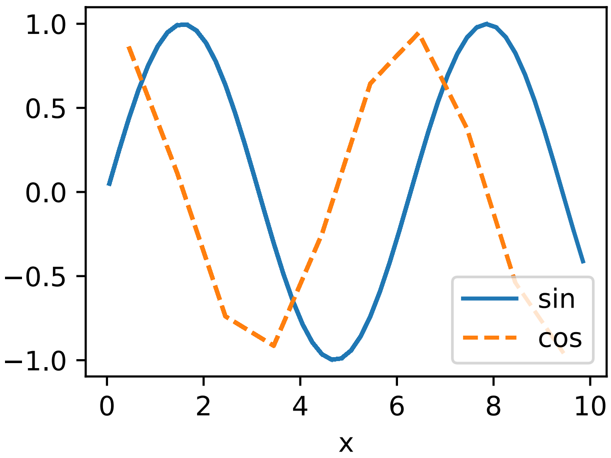
```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

```
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): # lr = learning rate
        super().__init__()
        self.save_hyperparameters()
        # weights are initialized to random numbers from a normal distribution
        # mean = 0, standard deviation = 0.1
        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):
    # Xw + b
    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):
    # squared error for loss function
    # y_hat are the predicted values
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

3.4.3. Defining the Optimization Algorithm

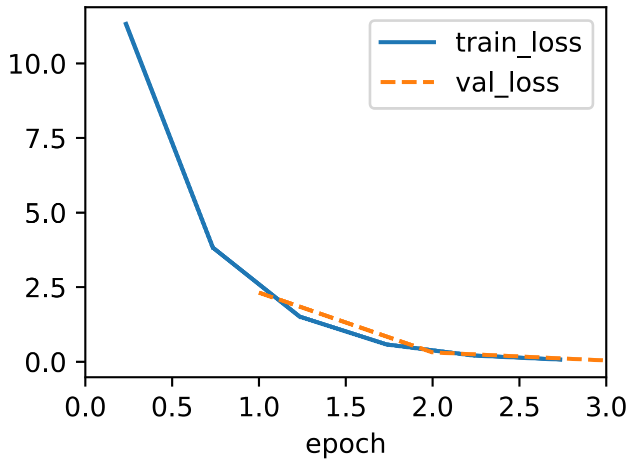
```
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        # step -> params are updated by -grad muliplied by learning rate
```

```
        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 an instance of the SGD class"""
    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()
    # loop throught each mini-batch
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:
                self.clip_gradients(self.gradient_clip_val, self.model)
            # update model parameters
            self.optim.step()
        self.train_batch_idx += 1
    # return if there is no validation data
    if self.val_dataloader is None:
        return
    self.model.eval()
    # validation loop
    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) # input_num = 2
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([ 0.0913, -0.1735])
error in estimating b: tensor([0.2198])
```

3. Discussions & Exercises

3.1. Linear Regression (Discussion)

Assumptions:

- relationship between features and target is approximately linear (the conditional mean can be expressed as a weighted sum of the features)
- data noise follows Gaussian distribution

Model:

- prediction $\hat{\mathbf{y}}$ can be expressed via the matrix–vector product where $\mathbf{X}$ is matrix of features, $\mathbf{w}$ are weights and b is a bias

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$$

Loss Function:

- to measure the quality (**fitness**) of a given model
- the distance between the real and predicted values
- for regression problems, the **squared error** is the most common loss function:

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

Prediction Problem and Optimal Solution:

- If the design matrix $\mathbf{X}$ has **full rank** (linearly independent columns) and one **extra column consisting of all 1** for the bias is added the problem can be written as:

$$\min \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$$

- optimal solution (analytic solution) is

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

Training (Optimizing) Model:

- iteratively reducing the error by updating the parameters in the direction that lowers the loss function
- the alogorithm is called **gradient descent**
- Gradient Descent Approches:
  - an average computed on every single example
  - single example at a time => stochastic gradient descent (SGD)
  - middleground => **Minibatch Stochastic Gradient Descent**
    - minimachtes, typically sized as powers of 2 (e.g. 32 - 256)

hyperparameters = parameters that are not updated in the training loop

3.2.6 Exercises

```
# 1. Locate full implementations of the above classes
# using ??d2l.HyperParameters.save_hyperparameters we can get the source
#Source:
def save_hyperparameters(self, ignore=[]):
    """Save function arguments into class attributes.

    Defined in :numref:`sec_utils`"""
    frame = inspect.currentframe().f_back
    _, _, _, local_vars = inspect.getargvalues(frame)
    self.hparams = {k:v for k, v in local_vars.items()
                    if k not in set(ignore+['self']) and not k.startswith('_')}
    for k, v in self.hparams.items():
        setattr(self, k, v)


# 2. Remove the save_hyperparameters statement in the B class
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        print(f"There is no self.a = {not hasattr(self, 'a')}, There is no self.b

b = B(a=1, b=2, c=3)
# after 'self.save_hyperparameters(ignore=['c'])' is removed the self.a, self.b
```

```
# and self.c are not initialized.
# save_hyperparameters initalizes parameters of def __init__(self, a, b, c)
# -> in this case a, b and c
# ignore=['c'] doesn't initalize parameters included in ignore list
```

```
There is no self.a = True, There is no self.b = True, There is no self.c = Tru
```

3.4 Linear Regression Training (Discussion)

- trainig steps:
    1. Initialize parameters - $\mathbf{w}$ and b
    2. Repeat until done
        - Compute gradient
        - update parameters

3.4.6. Exercises

```
# 1. What would happen if we were to initialize the weights to zero.
class LinearRegressionScratch(d2l.Module):
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01): # lr = learning rate
        super().__init__()
        self.save_hyperparameters()
        # weights are initialized to all zeros
        self.w = torch.zeros((num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)

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

Training the model with weights all initialized to **0**

```
model = LinearRegressionScratch(2, lr=0.03) # input_num = 2
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=5)
trainer.fit(model, data)
```

- There is no problem with training the model

```
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([ 0.0135, -0.0476])
error in estimating b: tensor([0.0359])
```

```
model.w, model.b
```

```
(tensor([[ 1.9846],
         [-3.3807]], requires_grad=True),
 tensor([4.1736], requires_grad=True))
```

```
data.w, data.b
```

```
(tensor([ 2.0000, -3.4000]), 4.2)
```

All Weights Initalized to 0s

- in simple linear regression problem the weights can be initalized to 0 and the model can still be trained
- in deep network models there would be a **symmetry problem**. Neurons in each layer are symmetric which means they would update in the exact same way if all weights would be 0

COSE474-2024F: DEEP LEARNING HW1

4. Linear Neural Networks for Classification

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

Show hidden output

4.2. The Image Classification Dataset

```
%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))
print(f"Number of images (10 categories) in dataset for traning: {len(data.train)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-i
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-i
100%|██████████| 26421880/26421880 [00:02<00:00, 10906083.88it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/Fash

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-l
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-l
100%|██████████| 29515/29515 [00:00<00:00, 210300.54it/s]
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/Fash
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-im
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-im
100%|██████████| 4422102/4422102 [00:04<00:00, 1097375.58it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/Fashi

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-la
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-la
100%|██████████| 5148/5148 [00:00<00:00, 12708815.18it/s]
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/Fashi

Number of images (10 categories) in dataset for traning: 60000 and testing: 10
```

```
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)
```

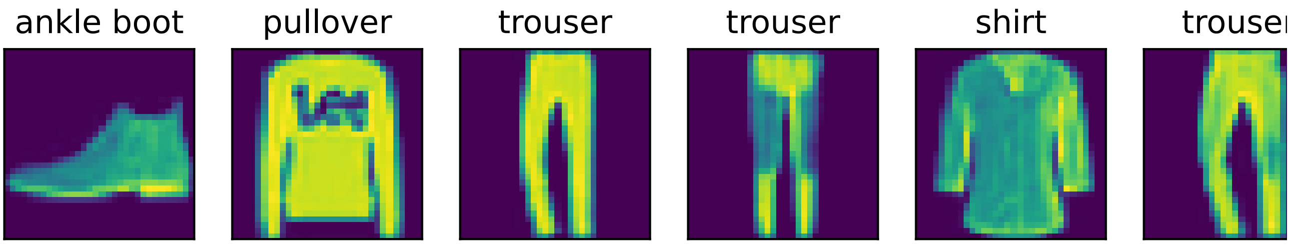
```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: Us
  warnings.warn(_create_warning_msg(
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

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

```
'13.42 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

```
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
```

4.4. Softmax Regression Implementation from Scratch

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.]]))
```

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

```
X = torch.rand((2, 5))
X
```

```
tensor([[0.8527, 0.4506, 0.6159, 0.4302, 0.5828],
        [0.4296, 0.5190, 0.2332, 0.6420, 0.9640]])
```

```
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2579, 0.1725, 0.2036, 0.1690, 0.1969],
         [0.1707, 0.1867, 0.1403, 0.2111, 0.2913]]),
 tensor([1., 1.]))
```

4.4.2. The Model

```
# flatten each image in the batch into a vector
# using reshape before passing the data through 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)
```

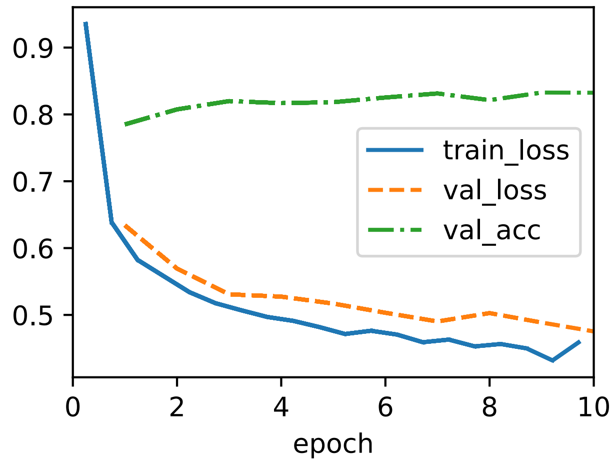
```
tensor(1.4979)
```

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

4.4.4. Training

```
@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
```

```
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)
```

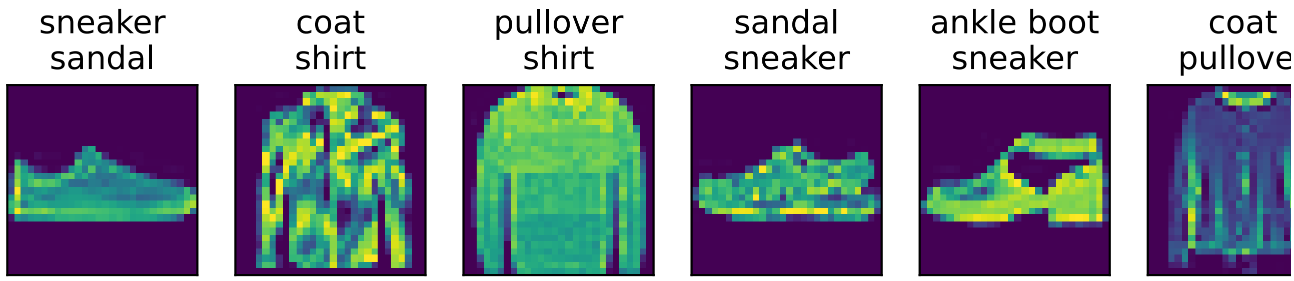


4.4.5. Prediction

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

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

```
# get the wrong predictions
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)
# first line = actual label, second line = prediction
```



4. Discussions & Exercises

4.1. Softmax Regression (Discussion)

- focuses on **classification problems**:
    - hard assignments of examples to categories (classes)

- soft assignments of examples to a probability that each category applies

Classification:

- a simple way to represent categorical data: **the one-hot encoding**
$$y \in \{(1,0,0),(0,1,0),(0,0,1)\}$$

Model:

- model is the same as in linear regression but there is an output for each category
$$\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Softmax:

- converts output into probabilities
- To avoid negative values, we use an exponential function
$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}), \text{ where } \hat{y}_i = \frac{e^{o_i}}{\sum_{j=1}^{n} e^{o_j}}$$

Loss Function:

- measures the difference between the predicted probability distribution and the true distribution
- in classification (with q classes) the loss function is called **Cross-Entropy Loss**:
$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log(\hat{y}_j)$$
- $\mathbf{y}$ is the true probability (**one-hot encoding**)

```
# Implementation of the Cross-Entropy loss function averaged over the batc
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()
```

4.2 Datasets (Discussion)

widely used datasets for image classification:

- **MNIST** - dataset of handwritten digits (too easy nowadays)
- **ImageNet** - a widely used and very large dataset of images
- **Fashion-MNIST** - released in 2017, 10 categories of clothing, resolution is 28x28

COSE474-2024F: DEEP LEARNING HW1

5. Multilayer Perceptrons

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

Show hidden output

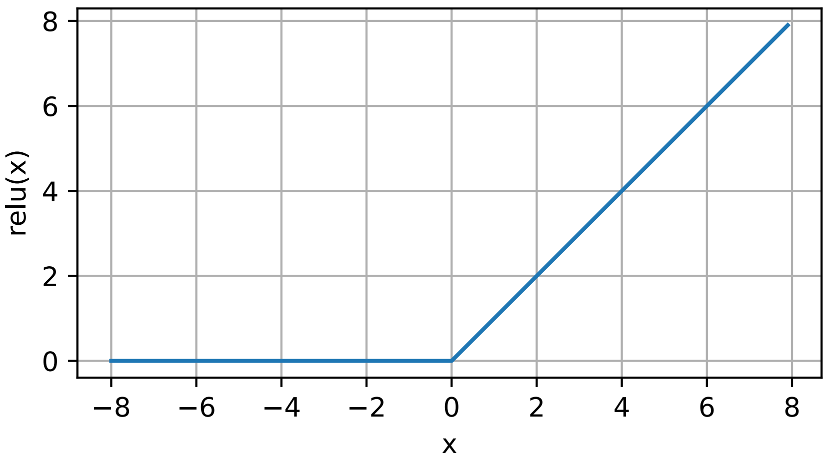
5.1. Multilayer Perceptrons

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

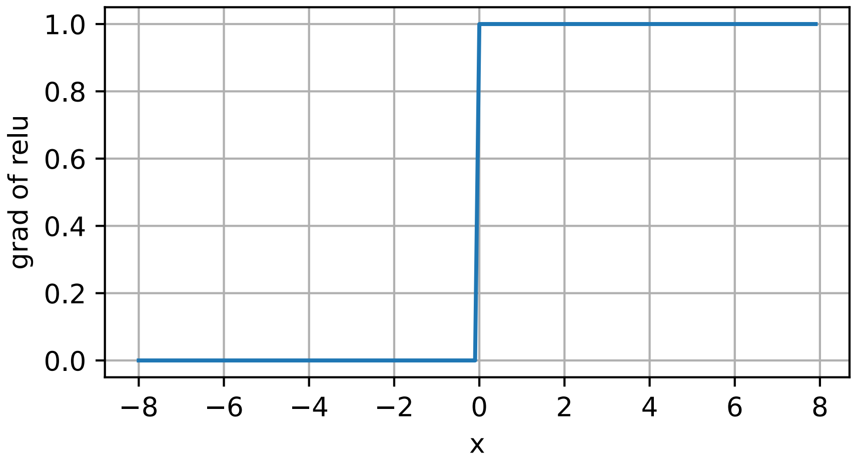
5.1.2. Activation Functions

5.1.2.1. ReLU Function

```
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))
```

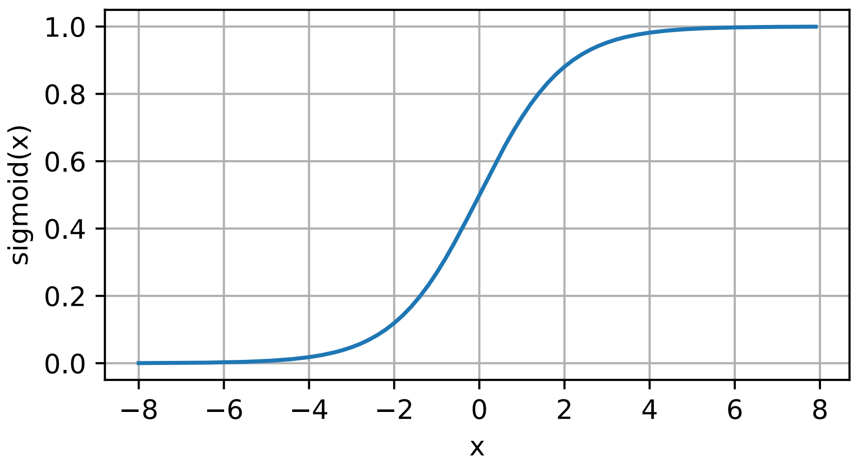



```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```

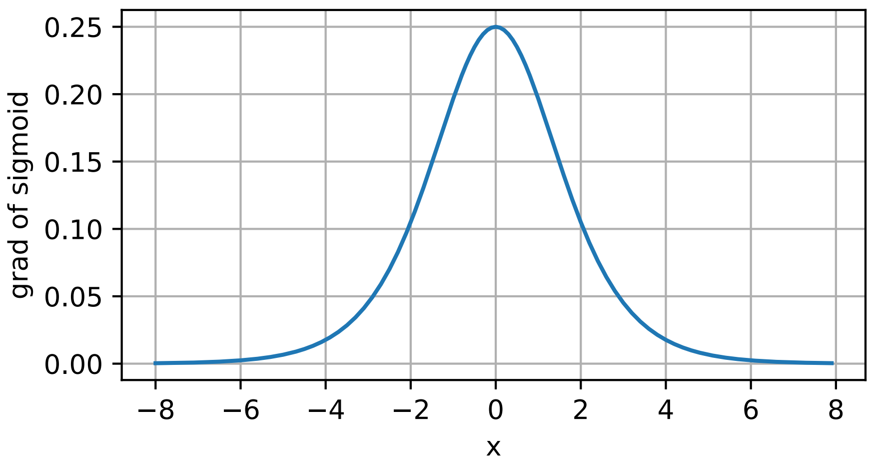


5.1.2.2. Sigmoid Function

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

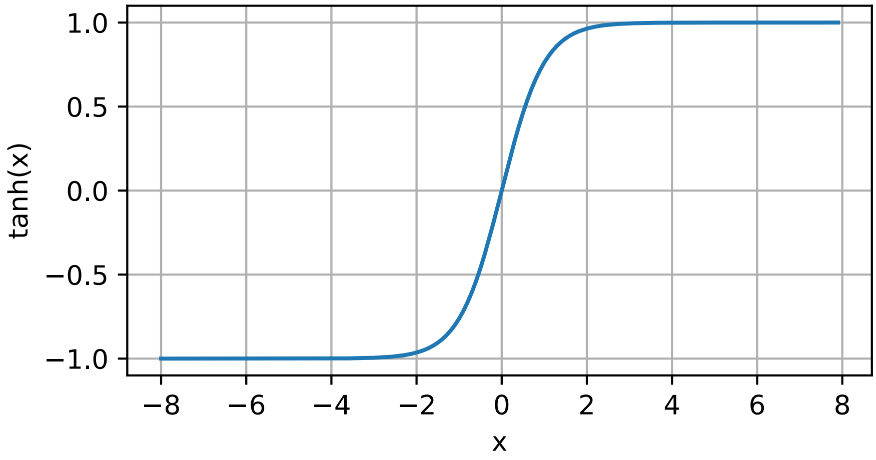



```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

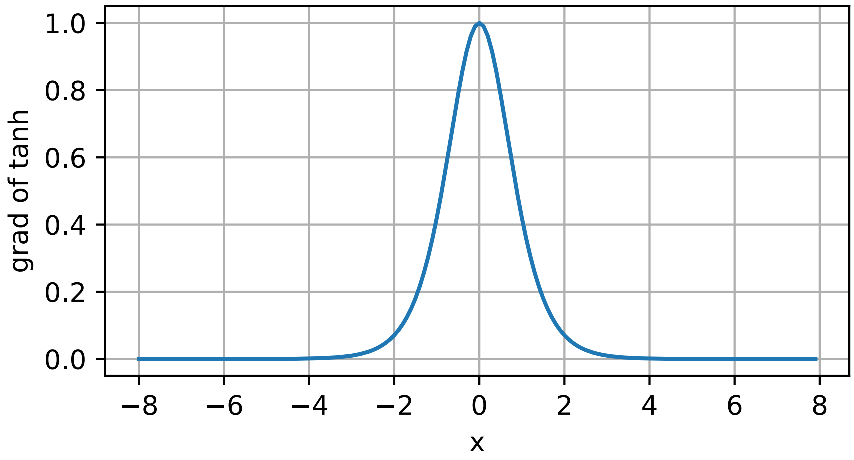


5.1.2.3. Tanh Function

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




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



5.1.1. Hidden Layers

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()
        # use nn.Parameter to automatically register a class attribute
        # as a parameter to be tracked by autograd
        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))
```

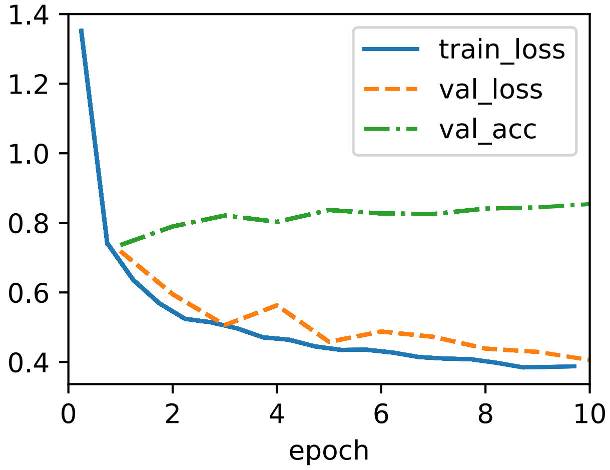
5.2.1.2. Model

```
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)
```



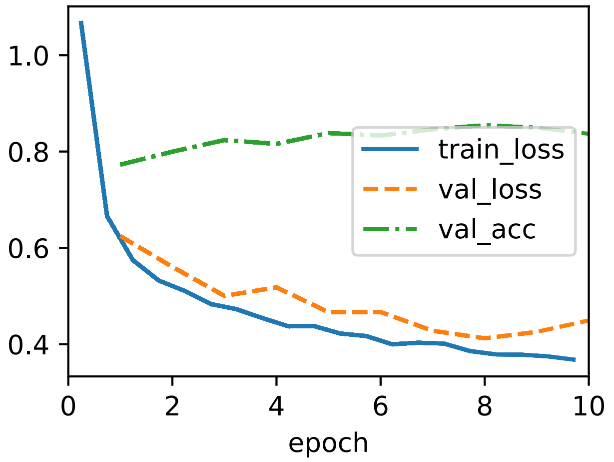
5.2.2. Concise Implementation

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. Discussions & Exercises

5.1 Multilayer Perceptrons (Discussion)

- by using activation function σ(**nonlinearity**) it is no longer possible to collapse MLP into a linear model:

$$\mathbf{H} = \sigma\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^1\right),$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^2$$

Modeling a Function:

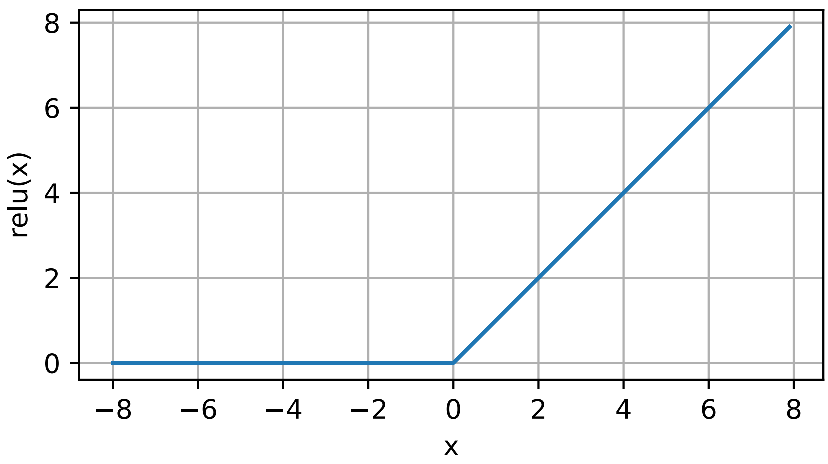
- In Cybenko (1989) for multilayer perceptrons (MLPs) and Micchelli (1984) for reproducing kernel Hilbert spaces (interpretable as radial basis function (RBF) networks), it is

suggested that even with a **single-hidden-layer** network, given enough nodes (possibly absurdly many), and the right set of weights, **we can model any function**.
- However, we can approximate many functions much more compactly by **using deeper** (rather than wider) networks (Simonyan and Zisserman, 2014)

Activation Functions:

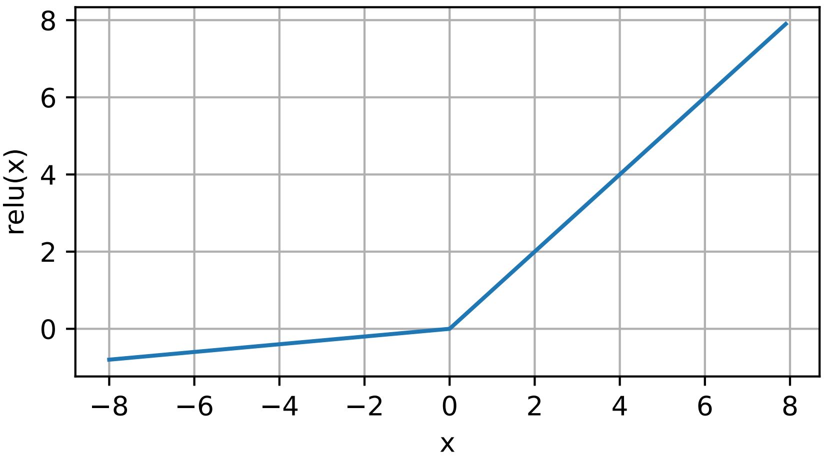
- **ReLU:**
  - the most popular activation function
  - $\mathrm{ReLU}(\mathrm{x}) = \max(\mathrm{x}, 0)$

```
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))
```



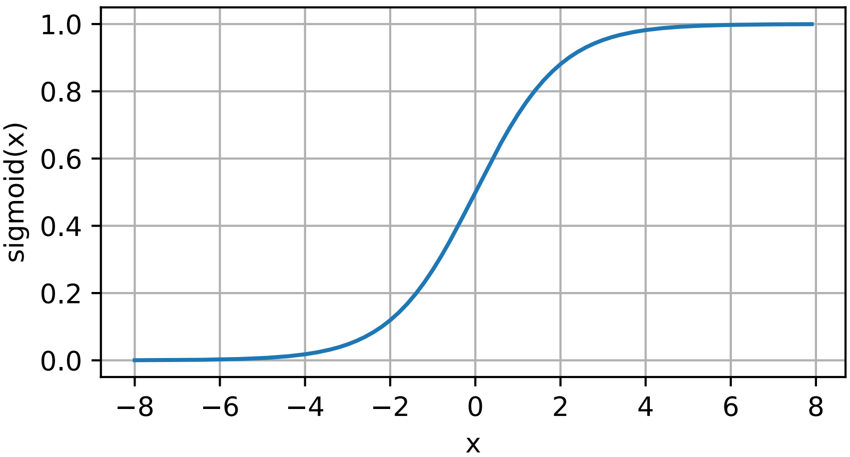
- **pReLU:**
  - $\mathrm{pReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases}$

```
alpha = 0.1
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.where(x >= 0, x, alpha * x)  # PReLU
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



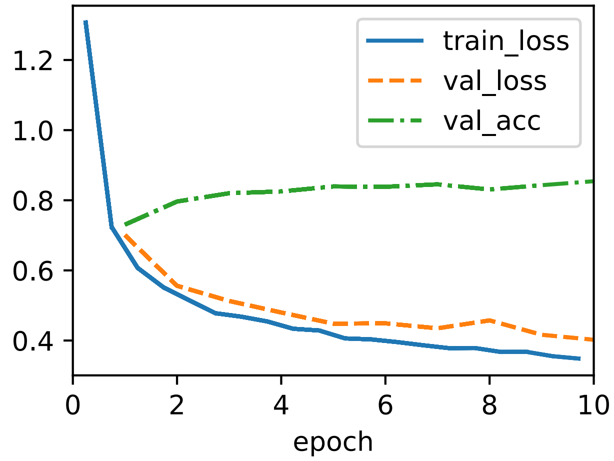
- **Sigmoid Function:**
  - $\text{sigmoid}(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

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



5.2.4 Exercises

```
# 2. Try adding a hidden layer to see how it affects the results.
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),
                                 # Extra hidden layer
                                 nn.ReLU(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))


model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
# The result seems to look quite similar to 1 hidden layer network
```



5.3.3. Backpropagation (Discussion)

+ Code + Text

- algorithm for calculating the gradient of neural network parameters
- J = **objective function** (model's regularized loss)
- The objective of backpropagation is to calculate the gradients:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^T + \lambda \mathbf{W}^{(2)}$$
$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^T + \lambda \mathbf{W}^{(1)}$$

- λ - hyperparameter
- $\mathbf{x}$ - input
- $\mathbf{h}$ - hidden activation vector
- $\mathbf{W}^{(1)}$ - weights of the hidden layer
- $\mathbf{W}^{(2)}$ - weights of the output layer
- $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$

5.3.4. Training Neural Networks (Discussion)

- once model parameters are initialized, we alternate **forward propagation** with **backpropagation**
- updating model parameters using gradients given by **backpropagation**
- backpropagation **reuses the stored intermediate values from forward propagation**
    - need to retain the intermediate values until backpropagation is complete