

✓ 2022320330 CS474 HW1

```
pip install d2l
```

 숨겨진 출력 표시

```
import torch
```

✓ 2.Preliminaries

✓ 2.1 Data Manipulation

- Two things to do with data : (i) acquire data, (ii) process them
- Use "tensor" class that has various advantages : (i) supports automatic differentiation, (ii) GPU leverage to accelerate numerical computation (NumPy only available for CPUs)

✓ 2.1.1 Getting Started

(1)

`arange(n, dtype)` : 0부터 n개의 숫자를 지정 간격, 지정 자료형으로 생성

`x.numel()` : x라는 텐서의 element수를 체크

`x.shape()` : x라는 텐서의 dimension 체크

```
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])`

(2)

`x.reshape(3,4)` : (12) 크기의 vector를 (3,4) 크기의 matrix로 반환. row별 rearrange

`x.reshape(-1,4)` : 3은 자동으로 계산 가능. 마지막 dim 크기만 지정하는 방식

`torch.zeros((2,3,4), dtype)` : 0으로 이루어진 dim=(2, 3, 4)의 텐서 생성

`torch.ones((2,3,4), dtype)` : 1로 이루어진 dim=(2,3,4)의 텐서 생성

`torch.randn(3,4)` : N(0,1)에서 난수 생성, (3,4)크기의 mat에 저장. `torch.randint` 등등 옵션 많음

`torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4,3,2,1]])` : tensor를 직접 지정 element로 빌드하는 코드.

`torch.from_numpy()` : numpy 배열과 tensor 배열 간의 자료형 전환


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

```
Y=x.reshape(-1,4)
```

```
print(X.shape, Y.shape)
```

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


```
print(torch.zeros((2,3,4), dtype=torch.float32), torch.ones((2,3,4), dtype=torch.float32), sep='\n')
```

 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.]]]])
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.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

(1)

`X[-1]` : 첫번째 dim의 마지막 원소 반환. (3,4) 크기일 경우 3번째 row 반환


`X[1:3]` : 첫번째 dim의 1,2번째 row 반환

`X[1,2]` : (1,2)번째 원소 반환.


`X[:, :]` : 첫번째 dim은 2번 이전까지, 두번째 dim은 모두 반환

단순 접근 뿐만 아니라, 값 변형도 가능함. 다만, 같은 메모리를 공유하는 텐서들은 모두 변형 적용됨.


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

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

▼ 2.1.3 Operations

(1)

practice elementwise operation with tensors

`torch.exp(x), torch.log(x)` : e, log 함수. elementwise 연산을 진행

`x+y, x-y, x*y, x/y, x**y` : 사칙연산 및 제곱. elementwise 연산을 진행

`x==y, x>y, x<y` : logical operation. elementwise 연산을 진행

`torch.exp(x)`, `torch.log(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]]),
    tensor([2.4849, 2.4849, 2.4849, 2.4849, 2.4849, 2.4849, 2.4849, 2.4849, 2.0794,
          2.1972, 2.3026, 2.3979]))
```

```
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]])
```

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

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

(2) concatenate

`torch.cat((T1, T2), dim)` : 지정한 `dim`에 대해 T1과 T2를 결합

```
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.])))
```

(3) tensor to scalar operation

`X.sum`, `X.mean`, `torch.max(X)`, `torch.min(X)`

`X.sum()`, `X.mean()`, `torch.max(X)`, `torch.min(X)`

```
⇒ (tensor(66.), tensor(5.5000), tensor(11.), tensor(0.))
```

✓ 2.1.4 Broadcasting

(1)

사이즈가 다른 텐서끼리 연산을 할 때, 자동으로 `size`를 맞춰서 연산을 함. `size`가 맞아떨어지게 적절한 값을 복사하여 `tensor` 확장

```
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

(1)

$Y = X + Y$:

- Y가 새로운 메모리에 할당됨.
- sum을 통한 parameter 업데이트 시 메모리 할당 횟수때문에 비효율적
- 다른 메모리를 가지는 동일한 파라미터를 모두 업데이트해야하기에 위험성이 있음

$Y[:] = X + Y$:

- in-place operation. 새로운 메모리 할당 안됨

$Y += X$

- 마찬가지로 in-place operation.

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

```
→ id(Z): 139195330241584
   id(Z): 139195373418784
```

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

```
→ id(Z): 139195330237984
   id(Z): 139195330237984
```

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

```
→ False
```

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

```
→ True
```

✓ 2.1.6 Conversion to Other Python Objects

(1) NumPy <-> torch

`T.numpy()` : pytorch tensor인 T를 numpy로 바꿈

`torch.from_numpy()` : numpy array를 pytorch tensor로 바꿈

```
A = X.numpy()
B = torch.from_numpy(A)
```

`type(A), type(B)`

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

(2) Tensor -> Scalar

`T.item()` : size 1 tensor인 `T`를 scalar로 바꿈

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

↔ (tensor([3.5000]), 3.5, 3.5, 3)

✓ 2.2 Data Preprocessing

Transform real-world data into processible form with pandas

✓ 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\n'
            'NA,NA,127500\n'
            '2,NA,106000\n'
            '4,Slate,178100\n'
            '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

(1) predict targets with inputs! If there exists NaN value or categorical feature,

Sol1) we can make dummy feature whose type is boolean.

ex) RoofType : Slate, NaN, ... -> RoofType_Slate = (T,F) / RoofType_NaN = (T,F)

Sol2) Or, we can imputate the missing values with mean

```
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

we can convert DataFrame into Tensor, enabling data to be fed into model.

```
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, x[2]

↔ (tensor([0, 1, 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]])
```

we can transpose the matrix as follows

```
A.T

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

Symmetric matrices satisfies following properties

```
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

`T.clone()` : T와 원소는 동일하지만 다른 메모리에 할당된 텐서

(1) Elementwise Sum, Elementwise Product (Hadamard product)

```
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.]])
```

(2) Scalar multiplication

```
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

(1) Sum of tensor

```
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.))
```

(2) Sum of tensor with specifying the reduced axes

`axis = 0` : row 방향으로 reduce(row dimension이 없어짐). column별 sum.

`axis = 1` : col 방향으로 reduce(col dimension이 없어짐). row별 sum.

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

```
↔ (tensor([[0., 1., 2.],
          [3., 4., 5.]]),
   tensor([3., 5., 7.]),
   tensor([ 3., 12.]),
   tensor(15.))
```

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

```
↔ tensor(True)
```

(3) mean

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

```
⇒ tensor(True)
```

(4) mean with specifying the reduced axes

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

```
⇒ tensor([True, True, True])
```

✓ 2.3.7 Non-Reduction Sum

Sometimes it is useful to keep the number of axes unchanged after operation.

Reducing operation eliminates the axis used for calculation.

(1) keepdims

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

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

```
A, sum_A, A / sum_A
```

```
⇒ (tensor([[0., 1., 2.],
           [3., 4., 5.]]),
   tensor([[ 3.],
           [12.]]),
   tensor([[0.0000, 0.3333, 0.6667],
           [0.2500, 0.3333, 0.4167]]))
```

(2) cumsum

```
A.cumsum(axis=0)
A.cumsum(axis=1)
```

```
⇒ tensor([[ 0.,  1.,  3.],
           [ 3.,  7., 12.]])
```

✓ 2.3.8 Dot Products

`torch.dot(x,y)` : equivalent operation with `torch.sum(x * y)`, `x.dot(y)`

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

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

✓ 2.3.9 Matrix-Vector Products

`torch.mv(A, x)` : equivalent operation with `A@x` (matrix & vector)

```
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

`torch.mm(A, B)` : equivalent operation with `A@B` (matrix & matrix)

```
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

(1) l2 norm

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

```
↔ tensor(5.)
```

(2) l1 norm

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

```
↔ tensor(7.)
```

(3) Frobenius norm

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

```
↔ tensor(6.)
```

✓ 2.5 Automatic Differentiation

- As we pass data through each successive function, the framework builds a computational graph that tracks how each value depends on others.
- When calculating derivatives, automatic differentiation works via backprop alg.

✓ 2.5.1 A Simple Function

assume that we are interested in differentiating the function

$$y = 2x^T x$$

w.r.t x .

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

```
↔ tensor([0., 1., 2., 3.])
```

(1) attach gradient attribute to tensor. For each tensor, we can enable "gradient storing" function

```
x.requires_grad_(True) # or, x = torch.arange(4.0, requires_grad = True)
x.grad
```

(2) forward pass

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

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

(3) backward pass

$$\frac{d}{dx}y = \frac{d}{dx}x^T x = 4x.$$

If we backpropagate gradient, then dy/dx is stored to $x.grad$

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

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

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

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

(4) test with another function

note that, the gradient is not automatically reset in the gradient buffer. New gradient is added to the already-stored gradient

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

```
↔ tensor([1., 1., 1., 1.])
```

✓ 2.5.2 Backward for Non-Scalar Variables

Now, consider y which is not a scalar. The result of computation is not y .

When calculating derivative w.r.t x , the result is a matrix called the Jacobian.

While Jacobian takes crucial role in advanced machine learning techniques (Normalizing Flows), we typically use the "sum" of the jacobians (not a full sum, but partial sum along dim). We do not need the original form of Jacobian in many cases

- When calculating loss value along batch dim and store loss value for each batch item in vector
- When updating parameters.

We should provide some vector v to tell PyTorch how to reduce the Jacobian. we want to make the function compute $v^T d_x y$ rather than $d_x y$. This vector v is called gradient.

```
x.grad.zero_()  
y = x * x #Hadamard multiplication  
y.backward(gradient=torch.ones(len(y))) # in x, 1^T d_x(y) is stored.  
x.grad
```

```
↔ tensor([0., 2., 4., 6.])
```

✓ 2.5.3 Detaching Computation

There may be some situations that we want to prevent gradient flow. (ex: GAN optimization).

Suppose that $z = x * y$ and $y = x * x$. When we want to examine the direct influence from x to z , neglecting the influence from x to y to z , we can detach the node for y from computational graph.

(1) detach y . y is considered constant

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

```
↔ tensor([0., 1., 4., 9.])
```

(2) non-detach. $z = y * x = x * x * x$ thus $dz/dx = 3x^2$.

```

x.grad.zero_()
y = x * x
z = y * x

z.sum().backward()
x.grad

↔ tensor([ 0.,  3., 12., 27.])

```

✓ 2.5.4 Gradients and Python Control Flow

computational graph enables backpropagation, even when there's logical gate between variables.

It is compatible with if-else, or for

```

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(2048.), tensor(2048., grad_fn=<DivBackward0>))

```

✓ 3. Linear Neural Networks for Regression

✓ 3.1 Linear Regression

```

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

```

✓ 3.1.1 Basics

(1) Assumptions of linear regression

- relationship between features x and target y is approximately linear
- conditional mean $E[Y|X = x]$ can be expressed as a weighted sum of the features x .
- noise follows Gaussian distribution

$$y = w^T x + b + \epsilon \text{ where } \epsilon \sim N(0, 1)$$

$$E[Y|X = x] = w^T x + b$$

(2) estimation using design matrix

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

(3) Loss Function for optimization

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

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(w, b)$$

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

(4) Analytic Solution

$$d\|y - Xw\|^2/dw = 2X^T(Xw - y) = 0 \rightarrow X^T y = X^T Xw$$

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

(5) Numerical optimization : Minibatch SGD

$$(w, b) \leftarrow (w, b) - \frac{\eta}{B} \frac{dL(w, b)}{d(w, b)}$$

✓ 3.1.2 Vectorization for Speed

for loop vs parallelization?

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
c = torch.zeros(n)
t = time.time()
```

(1) for loop

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

➡ '0.32764 sec'

(2) parallel computing

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

➡ '0.00030 sec'

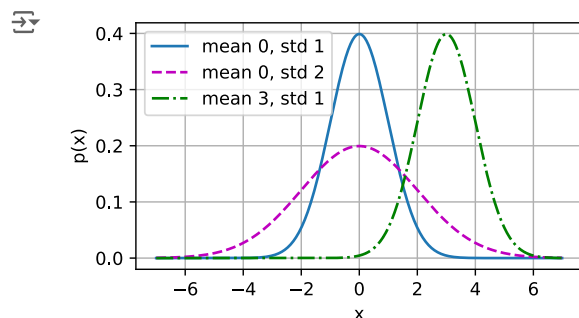
✓ 3.1.3 The Normal Distribution and Squared Loss

(1) Normal distribution

```
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])
```



(2) optimization of linear regression : Maximum Likelihood pov

Let $y = w^T x + b + \epsilon$ where $\epsilon \sim N(0, \sigma^2)$,

$$P(y|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - w^T x - b)^2\right)$$

In a likelihood p.o.v with dataset X ,

$$P(y|X) = \prod_{i=1}^n p(y^{(i)}|x^{(i)}),$$

$$-\log P(y|X) = \sum \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} (y^{(i)} - w^T x^{(i)} - b)^2$$

Just consider the squared term, and we can minimize the MSE.

However, derivation of MSE is done under several assumptions that σ is fixed, noise are independant, etc. If real data shows discrepancy with assumptions, the prediction might not be accurate.

✓ 3.1.4 Linear Regression as a Neural Network

perceptron can be mathematically modeled as follows :

$$y = \sum_i x_i w_i + b$$

✓ 3.2 Object-Oriented Design for Implementation

For regression, there are many components including the data, model, loss function, optimization algorithm.

We can treat each components as objects, thus should define classes.

We expect three classes :

- (i) Module that includes models, losses, optimization methods
- (ii) DataModule for providing data, dataloader
- (iii) Trainer that can combine (i) and (ii), for training process.

We should be able to design the framework, following the style of object-oriented programming

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

✓ 3.2.1 Utilites

(1) register function

register functions as methods of a class after the class has been created

```
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()
```

Decorate the method with `add_to_class(A)` func. It returns wrapper function, which wraps def do & class A

```
@add_to_class(A)
def do(self):
```

```
print('Class attribute "b" is', self.b)
```

```
a.do()
```

```
↔ Class attribute "b" is 1
```

(2) saving all arguments in a `__init__` method as class attributes.

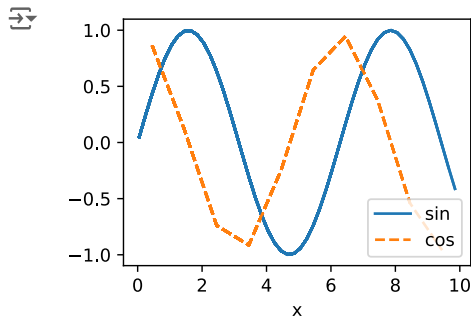
```
# Call the fully implemented HyperParameters class saved in d2l
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))
```

```
b = B(a=1, b=2, c=3)
```

```
↔ self.a = 1 self.b = 2
   There is no self.c = True
```

(3) ProgressBoard.

```
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

Module is a subclass of `nn.Module`, which provides convenient features for handling neural networks.

- if defining forward method, we can call this method by `a(X)`. built-in `_call_` method enables this.

```
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):
        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 initied'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / W
            self.trainer.num_train_batches
            n = self.trainer.num_train_batches / W
            self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / W
            self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
```

```

        ('train_' if train else 'val_') + key,
        every_n=int(n))

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

initialization, dataloader are integrated in DataModule

```

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

Implement the stochastic gradient descent from scratch!

```
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

1st. Initialize parameters (w,b)

2nd. Repeat until done

- Compute gradient
- Update Parameters

```
@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
```

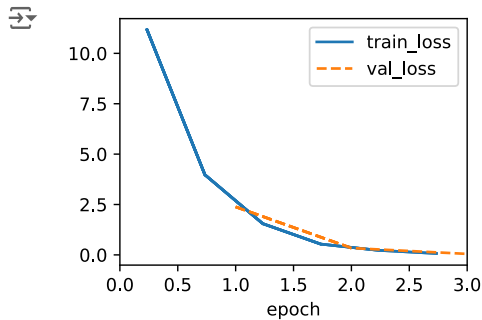


```

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([ 0.1093, -0.1835])
error in estimating b: tensor([0.2195])

```

✓ 4. Linear Neural Networks for Classification

✓ 4.1 Softmax Regression

✓ 4.1.1 Classification

(1) one - hot encoding

instead of predicting cat, chicken, dog classes we can predict $[y_1, y_2, y_3]$ where $[1,0,0]$ is cat, $[0,1,0]$ is chicken, $[0,0,1]$ is dog.

(2) Linear Model

Since the model should output multiple values, we can conduct multiple regressions, with model

$$y = Wx + b$$

(3) Softmax

Q. can we treat classification as a vector-valued regression problem? No. It has unsatisfactory in following ways :

- No guarantee that the outputs o_i sum up to 1
- No guarantee that outputs are nonnegative

A. Use softmax!

Final prediction is...

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

This idea dates back to Boltzmann. Refer to Energy-based models for using this point of view when describing problems in deep learning.

(4) Vectorization

We next parallelize the input & output as follows :

$$O = XW + b, \hat{Y} = \text{softmax}(O).$$

✓ 4.1.2 Loss Function

(1) Log-likelihood

we can interpret each output as probability, such as $\hat{y}_1 = P(y = cat|x)$.

Then, considering the whole dataset X for likelihood p.o.v, and assuming that each label is drawn independently from its respective distribution (i.i.d),

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

with log & cross-entropy loss,

$$-\log P(Y|X) = \sum_{i=1}^n -\log P(y^{(i)}|x^{(i)}) = \sum_{i=1}^n -y * \log \hat{y}, \text{ where } y \text{ is vector.}$$

(2) Softmax and Cross-Entropy Loss

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

$$d_{o_j} l(y, \hat{y}) = \text{softmax}(o)_j - y_j$$

The derivative of loss function is the difference between the probability by our model & true label. Every exponential family model, the gradients of the log-likelihood are given by precisely this term.

✓ 4.1.3 Information Theory Basics

(1) Entropy

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

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

in order to encode data drawn randomly from the distribution P , we need at least $H[P]$ "nats". "nat" is, equivalent of bit but when using base e rather than 2.

(2) Prediction and Entropy?

If we see a sequence of data, we are "surprised", and we gain "more information" if we observe an event with low probability. Thus, we can model "surprisal" as

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

Then, entropy is an "expected surprisal" of data, based on "true" distribution. If high entropy, high uncertainty.

(3) Cross-Entropy Revisited

cross-entropy from P to Q , i.e $H(P, Q)$, is expected surprisal of an observer with subjective probabilities Q upon observing data generated from P .

$$H(P, Q) = E_P[-\log(Q)]$$

The minimum is achieved when $P = Q$. In other words, $H(P, Q) \geq H(P)$. Intuitively, Cross Entropy can be interpreted as "expected surprisal when using the subjective belief system".

Thus, we can think of cross-entropy classification as

- maximizing the likelihood of the observed data
- minimizing surprisal required to communicate the labels

✓ 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))
len(data.train), len(data.val)
```

```
↔ (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'
```

```
↔ '13.75 sec'
```

✓ 4.2.3 Visualization

```
@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

Define a base class for classification models to simplify future code.

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

✓ 4.3.1 The Classifier Class

(1) Validation_step

- report both the loss value and the classification accuracy on a validation batch.
- draw an update for every num_val_batches batches. Not averaged along whole validation data

```
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)
```

(2) add SGD optimizer to Classifier

This is optional. We can separate the optimizer, or not.

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

✓ 4.3.2 Accuracy

Accuracy is computed as follows :

If y_{hat} is a matrix,

- we assume that the second dimension stores prediction scores for each class.
- Then, use `argmax` to obtain the predicted class by the index for the largest entry in each row
- Then, compare the predicted class with the ground truth y elementwise with equality operator
- Obtain result tensor contained with 0, or 1
- Take the Sum and obtain the number of correct predictions

```
@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

start with the mapping from scalars to probabilities. Recall the sum operator

```
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.])))
```

softmax requires

(i) exponentiation of each term (ii) a sum over each row to compute the normalization constant (iii) division of each row by its normalization constant

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

```
→ (tensor([[0.2263, 0.2492, 0.1302, 0.2485, 0.1457],
           [0.1371, 0.1828, 0.1545, 0.1994, 0.3262]]),
   tensor([1., 1.])))
```

✓ 4.4.2 The model

flatten 28 x 28 pixel images -> map to the probability of 10 classes

weights constitute a 784 x 10 matrix plus a 1 x 10 row vector for the biases.

```
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

cross entropy is the negative log likelihood of the predicted probability, assigned to the true label.

when y and y_{hat} , which is a label vector of two observed sample and prediction vector from model respectively, are given, we can calculate Cross Entropy easily with indexing.

```
y = torch.tensor([0, 2]) # first example : label is 0 / second example : label is 2
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y] # indexing. 0번째 데이터의 0번째 class에 대한 예측값, 1번째 데이터의 2번째 class에 대한 예측값
```

```
→ 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)
```

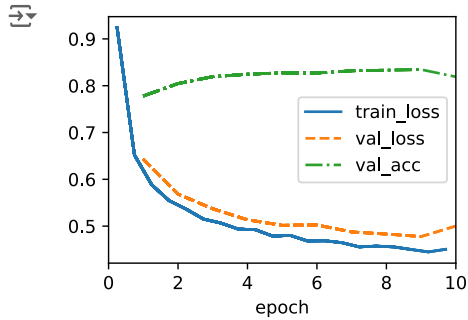
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
```

```
return cross_entropy(y_hat, y)
```

4.4.4 Training

hyperparameters : max_epochs, batch_size, lr

```
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

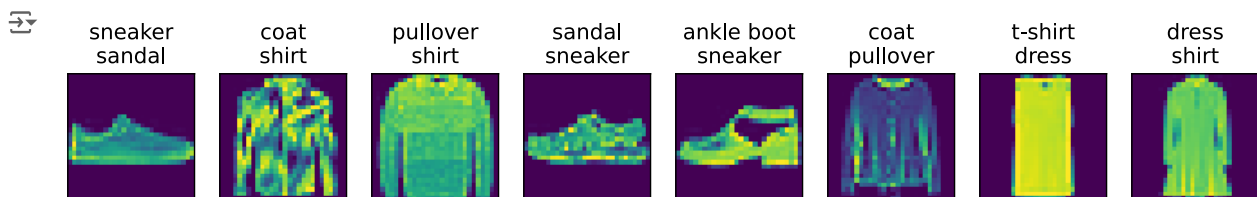
test the model with unseen data!

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

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

visualize the data incorrectly classified. first line of text output is actual labels, where second line the prediction.

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



5. Multilayer Perceptrons

5.1 Multilayer Perceptrons

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

5.1.1 Hidden Layers

regression is a strong assumption. Input and label doesn't necessarily have to be linear.

(1) Limitations of Linear Models

- Linearity assumes that, the increase in feature should always result to the increase in output (or vice versa).
- However, increasing the intensity of the pixel at location (13, 17) doesn't always increase the likelihood that the image depicts a dog.
- In other words, linear model differentiates cats & dogs, only by the brightness information.
- There might be "representations" of image which enables linear regression, but we don't know how to calculate it by hands.

To solve this problem of nonlinearity, there has been many methods : Decision trees, Kernel Methods, Brain-inspired network.

(2) Incorporating Hidden Layers and adding non-linearity.

Just stack many fully connected layers on top of one another. If there is L layers

- first $L - 1$ layers can be thought as "representation"
- final layer as linear predictor.

Then, incorporate the activation function between layers.

$$H = \sigma(XW^{(1)} + b^{(1)}), O = f(HW^{(2)} + b^{(2)})$$

(3) Universal Approximators

Q. How powerful a deep network could be?

A. According to Cybenko (1989) and Micchelli (1984), even with a single-hidden-layer network, any function can be approximated.

However, do not try to solve all of problems with one layer since kernel method is more effective in this framework.

Approximate many functions with deeper networks, not wider networks. (Simonyan and Zisserman, 2014).

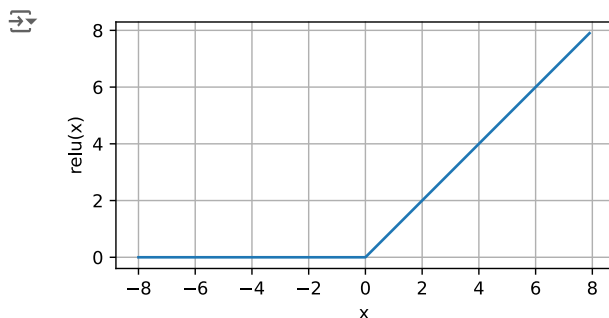
✓ 5.1.2 Activation Functions

(1) ReLU (Rectified Linear Unit)

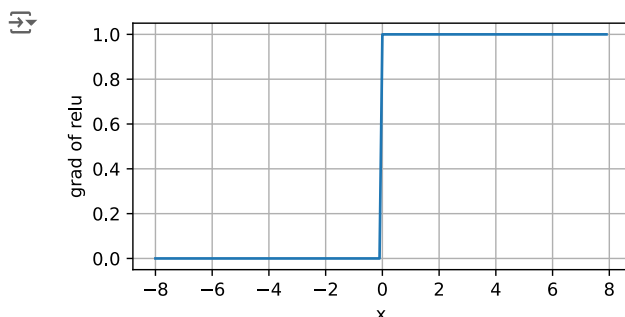
$$ReLU(x) = \max(x, 0)$$

ReLU retains only positive elements and discards all negative elements. If the value is exactly zero, take left-hand-side derivative and output as gradient. This is because we are not doing real mathematics.

```
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))
```



The gradient vanishes or becomes identical to the input. Due to the convenient, we often employ ReLU.

There are variations of ReLU function, such as

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

(2) Sigmoid Function

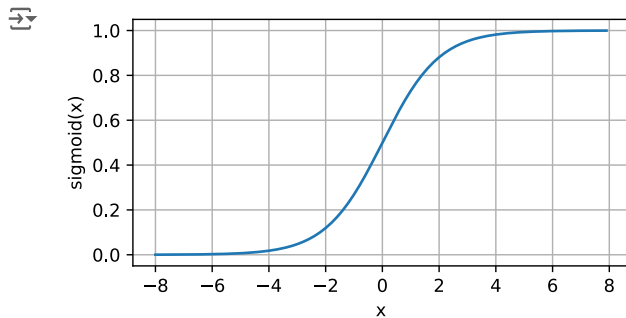
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

Early neural networks weren't based on gradient-based learning, thus employing the threshold function that outputs 0 or 1.

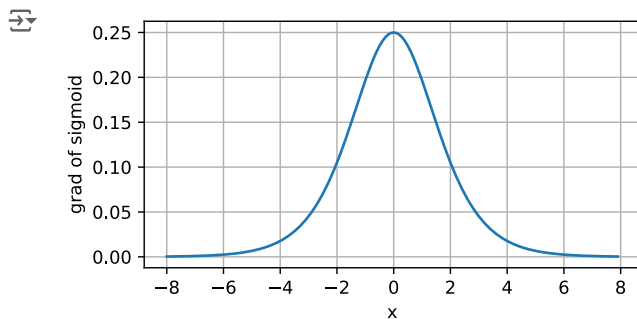
Sigmoids obtained attention for a while due to its similarity to the original threshold function, it was challenging for optimization. The gradient vanishes for large positive and large negative arguments.

Although replaced to ReLU now, sigmoid is still used.

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



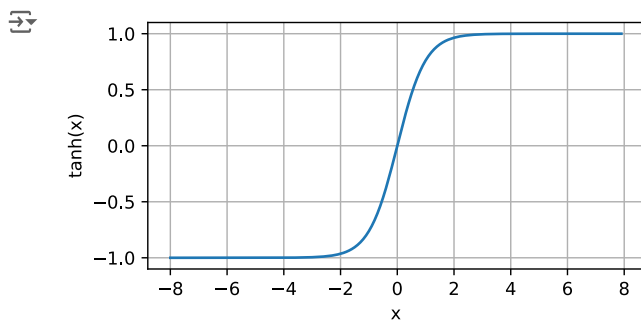
```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



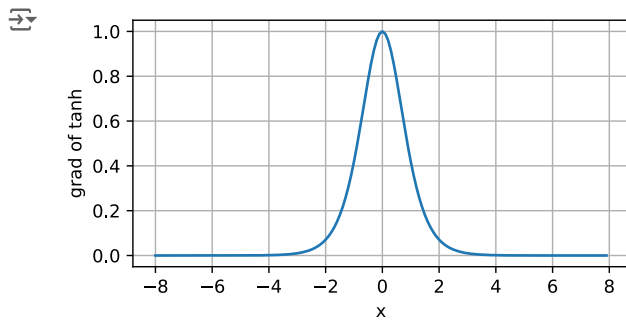
(3) Tanh Function

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

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




```
# 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 MLP

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

✓ 5.2.1 Implementation from Scratch

(1) Model

Do not consider the spatial structure among the pixels. Just flatten and feed them to MLP.

```
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))
```

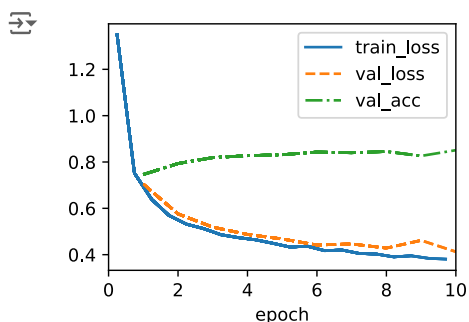
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

```
@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
```

(2) Training

Same as for softmax regression. Define model, data, trainer and fit.

```
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

(1) Model

We can use high-level APIs for building models.

`nn.Sequential()` : layer의 class를 인자로 받음. 순서대로 넣어줘야 함

`nn.Flatten()` : matrix를 vector로 평탄화

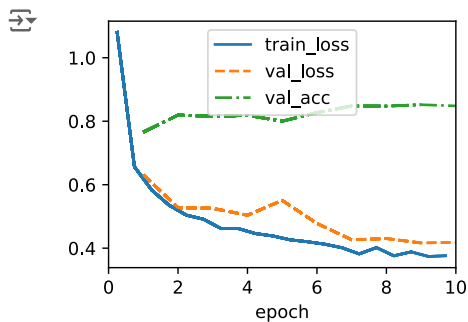
`nn.Linear()` : input 길이 상관 없이 output을 num_hidden으로 바꾸는건가? 아무튼 linear operation

`nn.ReLU()` : activation function

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

(2) Training

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



✓ 5.3 Forward Propagation, Backward Propagation, and Computational Graphs

Learn detailed mechanism of backpropagation, focusing on one-hidden-layer MLP with weight decay.

✓ 5.3.1 Forward Propagation

Let

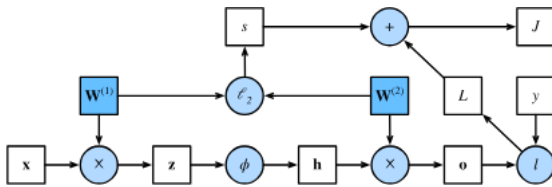
$$\begin{aligned}z &= W^{(1)}x \\h &= \Phi(z) \\o &= W^{(2)}h \\L &= l(o, y) \\s &= \frac{\lambda}{2}(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2) \\J &= L + s\end{aligned}$$

Then, J is called "objective function" of optimization.

Note that s is regularization term given the hyperparameter λ .

✓ 5.3.2 Computational Graph of Forward Propagation

operations and functions are intermediate node, where input and outputs are start & local end.



5.3.3 Backpropagation

(1) basic methods

Traverse the model in reverse order, and calculate the gradient based on chain rule. Note that,

$$\frac{dZ}{dX} = \text{prod}\left(\frac{dZ}{dY}, \frac{dY}{dX}\right)$$

(2) Backpropagate the above chain

- $\frac{dJ}{dL} = \frac{dJ}{ds} = 1$
- $\frac{dJ}{do} = \text{prod}\left(\frac{dJ}{dL}, \frac{dL}{do}\right) = \frac{dL}{do} \in R^q$
- $\frac{ds}{dW^{(i)}} = \lambda W^{(i)}$
- $\frac{dJ}{dW^{(2)}} = \text{prod}\left(\frac{dJ}{do}, \frac{do}{dW^{(2)}}\right) + \text{prod}\left(\frac{dJ}{ds}, \frac{ds}{dW^{(2)}}\right) = \frac{dJ}{do} h^T$
- $\frac{dJ}{dh} = \text{prod}\left(\frac{dJ}{do}, \frac{do}{dh}\right) = W^{(2)T} \frac{dJ}{do}$
- $\frac{dJ}{dz} = \text{prod}\left(\frac{dJ}{dh}, \frac{dh}{dz}\right) = \frac{dJ}{dh} * \Phi'(z)$
- $\frac{dJ}{dW^{(1)}} = \text{prod}\left(\frac{dJ}{dz}, \frac{dz}{dW^{(1)}}\right) + \text{prod}\left(\frac{dJ}{ds}, \frac{ds}{dW^{(1)}}\right) = \frac{dJ}{dz} x^T$

5.3.4 Training Neural Networks

(1) When Forward

- all the variables are calculated along the computational graph
- The local gradients are also saved to every local inputs

(2) When Backprop

- calculate the gradient "flow" from the final output (reverse)