

✓ Ch.0 Installation

✓ 0.1. Installation

```
1 !pip3 install d2l==1.0.3
```

 숨겨진 출력 표시


✓ Ch.2 Preliminaries

✓ 2.1. Data Manipulation

✓ 2.1.1. Getting Started

```
1 import torch
2
```

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

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

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

 12


```
1 len(x)
```

 12


```
1 x.shape
```

 torch.Size([12])

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

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


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

 tensor([[[[0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.]],
 [[0., 0., 0., 0.],
 [0., 0., 0., 0.],
 [0., 0., 0., 0.]])])

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

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

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

 tensor([[0.1192, -0.7301, 0.9975, -0.9678],
 [-0.3968, -0.7179, 0.2225, -0.7805],
 [-0.1048, -0.4151, -1.9217, 0.8286]])

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

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

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

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

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

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

2.1.3. Operations

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

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

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

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

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

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

```
1 X==Y
```

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

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

```
→ tensor(66.)
```

2.1.4. Broadcasting

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

```

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

```

```
1 a+b
```

```

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

```

2.1.5. Saving Memory

```

1 before=id(Y)
2 Y=Y+X
3 id(Y)==before

```

```
False
```

```

1 Z=torch.zeros_like(Y)
2 print('id(Z):', id(Z))
3 Z[:]=X+Y
4 print('id(Z):', id(Z))

```

```

id(Z): 132633704226048
id(Z): 132633704226048

```

```

1 before=id(X)
2 X+=Y
3 id(X)==before

```

```
True
```

2.1.6. Conversion to Other Python Objects

```

1 A=X.numpy()
2 B=torch.from_numpy(A)
3 type(A), type(B)

```

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

```

1 a=torch.tensor([3.5])
2 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

```
1 import os
```

```

1 os.makedirs(os.path.join('.', 'data'), exist_ok=True)
2 data_file=os.path.join('.', 'data', 'house_tiny.csv')
3 with open(data_file, 'w') as f:
4     f.write('NumRooms,RoofType,Price\n')
5     f.write('NA,NA,127500\n')
6     f.write('2,NA,106000\n')
7     f.write('4,Slate,178100\n')
8     f.write('NA,NA,140000\n')

```

```
1 import pandas as pd
```

```

1 data=pd.read_csv(data_file)
2 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 inputs, targets=data.iloc[:, 0:2], data.iloc[:, 2]
2 inputs=pd.get_dummies(inputs, dummy_na=True)
3 print(inputs)
```

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

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

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

2.2.3. Conversion to the Tensor Format

```
1 import torch
```

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

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

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

2.3.2. Vectors

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

```
↔ tensor([0, 1, 2])
```

```
1 x[2]
```

```
↔ tensor(2)
```

```
1 len(x)
```

```
↔ 3
```

```
1 x.shape
```

```
↔ torch.Size([3])
```

2.3.3. Matrices

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

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

```
1 A.T
```

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

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

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

2.3.4. Tensors

```
1 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

```
1 A=torch.arange(6, dtype=torch.float32).reshape(2, 3)
2 B=A.clone()
3 A, A+B
```

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

```
1 A*B
```

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

```
1 a=2
2 X=torch.arange(24).reshape(2, 3, 4)
3 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 x=torch.arange(3, dtype=torch.float32)
2 x, x.sum()
```

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

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

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

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

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

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

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

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

```
↔ tensor(True)
```

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

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

```
1 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

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

```
2 sum_A, sum_A.shape
```

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

```
1 A/sum_A
```

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

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

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

✓ 2.3.8. Dot Products

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

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

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

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

```
↔ tensor(3.)
```

✓ 2.3.9. Matrix-Vector Products

```
1 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

```
1 B=torch.ones(3, 4)
```

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

```
2 torch.norm(u)
```

```
↔ tensor(5.)
```

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

```
↔ tensor(7.)
```

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

```
↔ tensor(6.)
```

✓ 2.5. Automatic Differentiation

```
1 import torch
```

✓ 2.5.1. A Simple Function

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

```
2 x
```

```
↔ tensor([0., 1., 2., 3.])
```

```
1 x.requires_grad_(True)
```

```
2 x.grad
```

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

```
2 y
```

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

```
1 y.backward()
```

```
2 x.grad
```

```
↔ tensor([ 0., 4., 8., 12.])
```

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

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

```
1 x.grad.zero_()
```

```
2 y=x.sum()
```

```
3 y.backward()
```

```
4 x.grad
```

```
↔ tensor([1., 1., 1., 1.])
```

✓ 2.5.2. Backward for Non-Scalar Variables

```
1 x.grad.zero_()
```

```
2 y=x*x
```

```
3 y.backward(gradient=torch.ones(len(y)))
```

```
4 x.grad
```

```
↔ tensor([0., 2., 4., 6.])
```

✓ 2.5.3. Detaching Computation

```
1 x.grad.zero_()
```

```
2 y=x*x
```

```
3 u=y.detach_()
```

```
4 z=u * x
```

```
5
```

```
6 z.sum().backward()
```

```
7 x.grad==u
```

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

```
1 x.grad.zero_()
```

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

```
3 x.grad==2 * x
```

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

2.5.4. Gradients and Python Control Flow

```

1 def f(a):
2     b=a*2
3     while b.norm()<1000:
4         b=b*2
5         if b.sum()>0:
6             c=b
7         else:
8             c=100*b
9     return c

1 a=torch.randn(size=(), requires_grad=True)
2 d=f(a)
3 d.backward()

1 a.grad==d/a

```

tensor(True)

Ch.3 Linear Neural Networks for Regression

3.1. Linear Regression

```

1 %matplotlib inline
2 import math
3 import time
4 import numpy as np
5 import torch
6 from d2l import torch as d2l

```

3.1.2. Vectorisation for Speed

```

1 n=10000
2 a=torch.ones(n)
3 b=torch.ones(n)

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

```

'0.16245 sec'

```

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

```

'0.00030 sec'

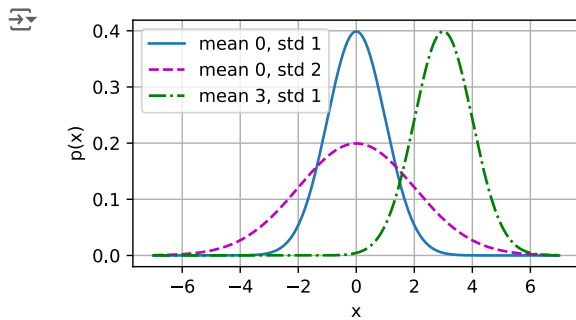
3.1.3. The Normal Distribution and Squared Loss

```

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

1 # Use NumPy again for visualization
2 x=np.arange(-7, 7, 0.01)
3
4 # Mean and standard deviation pairs
5 params=[(0, 1), (0, 2), (3, 1)]
6 d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
7           ylabel='p(x)', figsize=(4.5, 2.5),
8           legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])

```

✓ 3.2. Object-Oriented Design for Implementation

```
1 import time
2 import numpy as np
3 import torch
4 from torch import nn
5 from d2l import torch as d2l
```

✓ 3.2.1. Utilities

```
1 def add_to_class(Class):
2     def wrapper(obj):
3         setattr(Class, obj.__name__, obj)
4     return wrapper
```

```
1 class A:
2     def __init__(self):
3         self.b=1
4
5 a=A()
```

```
1 @add_to_class(A)
2 def do(self):
3     print('Class attribute "b" is', self.b)
4
5 a.do()
```

→ Class attribute "b" is 1

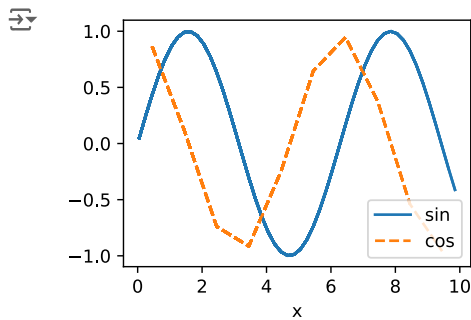
```
1 class HyperParameters:
2     def save_hyperparameters(self, ignore=[]):
3         raise NotImplemented
```

```
1 class B(d2l.HyperParameters):
2     def __init__(self, a, b, c):
3         self.save_hyperparameters(ignore=['c'])
4         print('self.a =', self.a, 'self.b =', self.b)
5         print('There is no self.c =', not hasattr(self, 'c'))
6
7 b=B(a=1, b=2, c=3)
```

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

```
1 class ProgressBoard(d2l.HyperParameters):
2     def __init__(self, xlabel=None, ylabel=None, xlim=None,
3                 ylim=None, xscale='linear', yscale='linear',
4                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2', 'C3'],
5                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
6         self.save_hyperparameters()
7
8     def draw(self, x, y, label, every_n=1):
9         raise NotImplemented
```

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



3.2.2. Models

```

1 class Module(nn.Module, d2l.HyperParameters):
2     def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
3         super().__init__()
4         self.save_hyperparameters()
5         self.board = ProgressBoard()
6
7     def loss(self, y_hat, y):
8         raise NotImplementedError
9
10    def forward(self, X):
11        assert hasattr(self, 'net'), 'Neural network is defined'
12        return self.net(X)
13
14    def plot(self, key, value, train):
15        assert hasattr(self, 'trainer'), 'Trainer is not initied'
16        self.board.xlabel='epoch'
17        if train:
18            x=self.trainer.train_batch_idx/W
19            self.trainer.num_train_batches
20            n=self.trainer.num_train_batches/W
21            self.plot_train_per_epoch
22        else:
23            x=self.trainer.epoch+1
24            n=self.trainer.num_val_batches/W
25            self.plot_valid_per_epoch
26        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
27                        ('train_' if train else 'val_')+key,
28                        every_n=int(n))
29
30    def training_step(self, batch):
31        l=self.loss(self(*batch[:-1]), batch[-1])
32        self.plot('loss', l, train=True)
33        return l
34
35    def validation_step(self, batch):
36        l=self.loss(self(*batch[:-1]), batch[-1])
37        self.plot('loss', l, train=False)
38
39    def configure_optimizers(self):
40        raise NotImplementedError

```

3.2.3. Data

```

1 class DataModule(d2l.HyperParameters):
2     def __init__(self, root='../data', num_workers=4):
3         self.save_hyperparameters()
4
5     def get_dataloader(self, train):
6         raise NotImplementedError
7
8     def train_dataloader(self):
9         return self.get_dataloader(train=True)
10
11    def val_dataloader(self):
12        return self.get_dataloader(train=False)

```

3.2.4. Training

```

1 class Trainer(d2l.HyperParameters):
2     def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):

```

```

3     self.save_hyperparameters()
4     assert num_gpus==0, 'No GPU support yet'
5
6     def prepare_data(self, data):
7         self.train_data_loader=data.train_data_loader()
8         self.val_data_loader=data.val_data_loader()
9         self.num_train_batches=len(self.train_data_loader)
10        self.num_val_batches=(len(self.val_data_loader)
11                               if self.val_data_loader is not None else 0)
12
13    def prepare_model(self, model):
14        model.trainer=self
15        model.board.xlim=[0, self.max_epochs]
16        self.model=model
17
18    def fit(self, model, data):
19        self.prepare_data(data)
20        self.prepare_model(model)
21        self.optim=model.configure_optimizers()
22        self.epoch=0
23        self.train_batch_idx=0
24        self.val_batch_idx=0
25        for self.epoch in range(self.max_epochs):
26            self.fit_epoch()
27
28    def fit_epoch(self):
29        raise NotImplementedError

```

✓ 3.4. Linear Regression Implementation from Scratch

```

1 %matplotlib inline
2 import torch
3 from d2l import torch as d2l

```

✓ 3.4.1. Defining the Model

```

1 class LinearRegressionScratch(d2l.Module):
2     def __init__(self, num_inputs, lr, sigma=0.01):
3         super().__init__()
4         self.save_hyperparameters()
5         self.w=torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
6         self.b=torch.zeros(1, requires_grad=True)

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

```

✓ 3.4.2. Defining the Loss Function

```

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

```

✓ 3.4.3. Defining the Optimization Algorithm

```

1 class SGD(d2l.HyperParameters):
2     def __init__(self, params, lr):
3         self.save_hyperparameters()
4
5     def step(self):
6         for param in self.params:
7             param-=self.lr*param.grad
8
9     def zero_grad(self):
10        for param in self.params:
11            if param.grad is not None:
12                param.grad.zero_()

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

```

3.4.4. Training

```

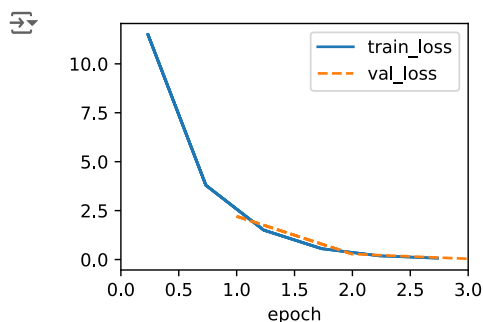
1 @d2l.add_to_class(d2l.Trainer)
2 def prepare_batch(self, batch):
3     return batch
4
5 @d2l.add_to_class(d2l.Trainer)
6 def fit_epoch(self):
7     self.model.train()
8     for batch in self.train_dataloader:
9         loss=self.model.training_step(self.prepare_batch(batch))
10        self.optim.zero_grad()
11        with torch.no_grad():
12            loss.backward()
13            if self.gradient_clip_val>0:
14                self.clip_gradients(self.gradient_clip_val, self.model)
15        self.optim.step()
16        self.train_batch_idx+=1
17    if self.val_dataloader is None:
18        return
19    self.model.eval()
20    for batch in self.val_dataloader:
21        with torch.no_grad():
22            self.model.validation_step(self.prepare_batch(batch))
23        self.val_batch_idx+=1

```

```

1 model=LinearRegressionScratch(2, lr=0.03)
2 data=d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
3 trainer=d2l.Trainer(max_epochs=3)
4 trainer.fit(model, data)

```



```

1 with torch.no_grad():
2     print(f'error in estimating w: {data.w-model.w.reshape(data.w.shape)}')
3     print(f'error in estimating b: {data.b-model.b}')

```

error in estimating w: tensor([0.1017, -0.1478])
error in estimating b: tensor([0.2090])

Ch.4 Linear Neural Networks for Classification

4.1. Softmax Regression

(No Codes)

4.2. The Image Classification Dataset

```

1 %matplotlib inline
2 import time
3 import torch
4 import torchvision
5 from torchvision import transforms
6 from d2l import torch as d2l
7
8 d2l.use_svg_display()

```

4.2.1. Loading the Dataset

```

1 class FashionMNIST(d2l.DataModule):
2     def __init__(self, batch_size=64, resize=(28, 28)):
3         super().__init__()
4         self.save_hyperparameters()
5         trans=transforms.Compose([transforms.Resize(resize),
6                                   transforms.ToTensor()])
7         self.train=torchvision.datasets.FashionMNIST(
8             root=self.root, train=True, transform=trans, download=True)
9         self.val=torchvision.datasets.FashionMNIST(
10            root=self.root, train=False, transform=trans, download=True)

```

```

1 data=FashionMNIST(resize=(32, 32))
2 len(data.train), len(data.val)

```

Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz>
 Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz> to ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz [100%|#####| 26421880/26421880 [00:04<00:00, 6104272.29it/s]
 Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz>
 Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz> to ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz [100%|#####| 29515/29515 [00:00<00:00, 342907.86it/s]
 Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz>
 Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz> to ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz [100%|#####| 4422102/4422102 [00:00<00:00, 6334215.26it/s]
 Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz>
 Downloading <http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz> to ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz [100%|#####| 5148/5148 [00:00<00:00, 5116653.32it/s]
 Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

(60000, 10000)

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

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

```

1 @d2l.add_to_class(FashionMNIST)
2 def text_labels(self, indices):
3     labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
4              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
5     return [labels[int(i)] for i in indices]

```

4.2.2. Reading a Minibatch

```

1 @d2l.add_to_class(FashionMNIST)
2 def get_dataloader(self, train):
3     data=self.train if train else self.val
4     return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
5                                         num_workers=self.num_workers)

```

```

1 X, y=next(iter(data.train_dataloader()))
2 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. This may cause OOM on GPU if you have limited memory. You can set torch.utils.data.num_workers=1 to reduce the number of worker processes to avoid OOM.
 warnings.warn(_create_warning_msg(
 torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64

```

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

```

```
'10.86 sec'
```

4.2.3. Visualisation

```

1 def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
2     raise NotImplementedError

```

```

1 @d2l.add_to_class(FashionMNIST)
2 def visualize(self, batch, nrows=1, ncols=8, labels=[]):
3     X, y=batch
4     if not labels:
5         labels=self.text_labels(y)
6     d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
7 batch=next(iter(data.val_dataloader()))
8 data.visualize(batch)

```



4.3. The Base Classification Model

```

1 import torch
2 from d2l import torch as d2l

```

4.3.1. The Classifier Class

```

1 class Classifier(d2l.Module):
2     def validation_step(self, batch):
3         Y_hat = self(*batch[:-1])
4         self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
5         self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)

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

```

4.3.2. Accuracy

```

1 @d2l.add_to_class(Classifier)
2 def accuracy(self, Y_hat, Y, averaged=True):
3     Y_hat=Y_hat.reshape((-1, Y_hat.shape[-1]))
4     preds=Y_hat.argmax(axis=1).type(Y.dtype)
5     compare=(preds==Y.reshape(-1)).type(torch.float32)
6     return compare.mean() if averaged else compare

```

4.4. Softmax Regression Implementation from Scratch

```

1 import torch
2 from d2l import torch as d2l

```

4.4.1. The Softmax

```

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

```

```

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

```

```

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

```

```

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

```

```

(tensor([[0.2813, 0.1796, 0.2314, 0.1488, 0.1589],
        [0.1986, 0.2282, 0.1475, 0.2254, 0.2003]]),
 tensor([[1.0000, 1.0000]]))

```

4.4.2. The Model

```

1 class SoftmaxRegressionScratch(d2l.Classifier):
2     def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
3         super().__init__()
4         self.save_hyperparameters()
5         self.W=torch.normal(0, sigma, size=(num_inputs, num_outputs),
6                                     requires_grad=True)
7         self.b=torch.zeros(num_outputs, requires_grad=True)
8
9     def parameters(self):
10        return [self.W, self.b]

```

```

1 @d2l.add_to_class(SoftmaxRegressionScratch)
2 def forward(self, X):
3     X=X.reshape((-1, self.W.shape[0]))
4     return softmax(torch.matmul(X, self.W)+self.b)

```

4.4.3. The Cross-Entropy Loss

```

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

```

```
→ tensor([0.1000, 0.5000])
```

```

1 def cross_entropy(y_hat, y):
2     return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()
3
4 cross_entropy(y_hat, y)

```

```
→ tensor(1.4979)
```

```

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

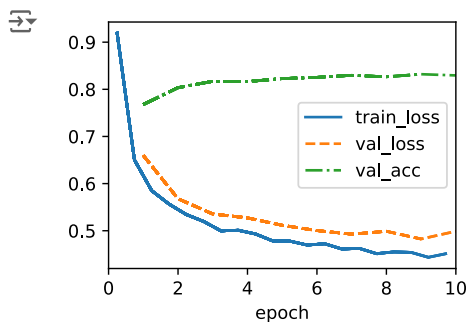
```

4.4.4. Training

```

1 data=d2l.FashionMNIST(batch_size=256)
2 model=SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
3 trainer=d2l.Trainer(max_epochs=10)
4 trainer.fit(model, data)

```



4.4.5. Prediction

```

1 X, y=next(iter(data.val_data_loader()))
2 preds=model(X).argmax(axis=1)
3 preds.shape

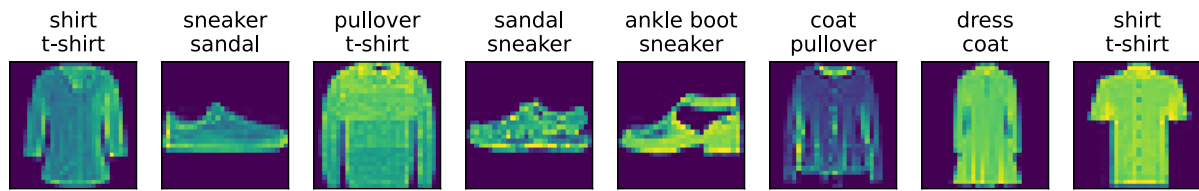
```

```
→ torch.Size([256])
```

```

1 wrong=preds.type(y.dtype)!=y
2 X, y, preds=X[wrong], y[wrong], preds[wrong]
3 labels=[a+'%0n'+b for a, b in zip(
4     data.text_labels(y), data.text_labels(preds))]
5 data.visualize([X, y], labels=labels)

```



✓ Ch.5 Multilayer Perceptrons

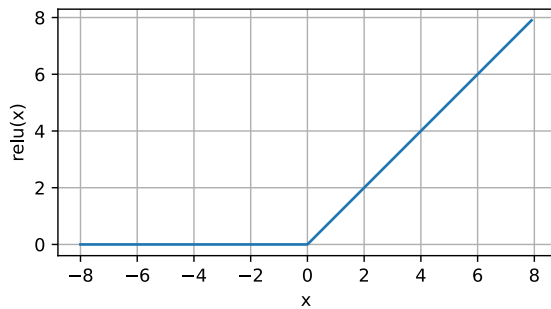
✓ 5.1. Multilayer Perceptrons

```
1 %matplotlib inline
2 import torch
3 from d2l import torch as d2l
```

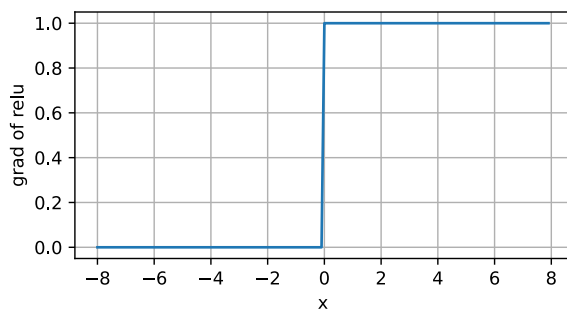
✓ 5.1.2. Activation Functions

✓ 5.1.2.1. ReLU Function

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

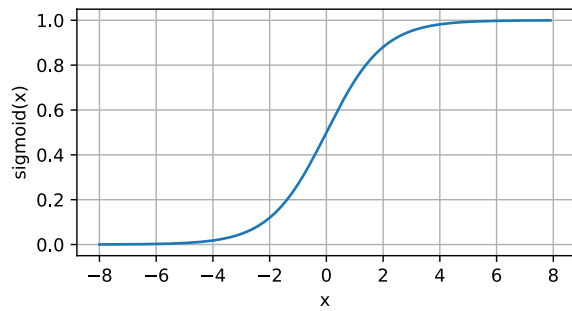


```
1 y.backward(torch.ones_like(x), retain_graph=True)
2 d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```

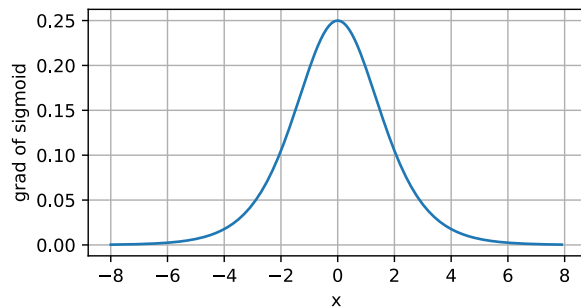


✓ 5.1.2.2. Sigmoid Function

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

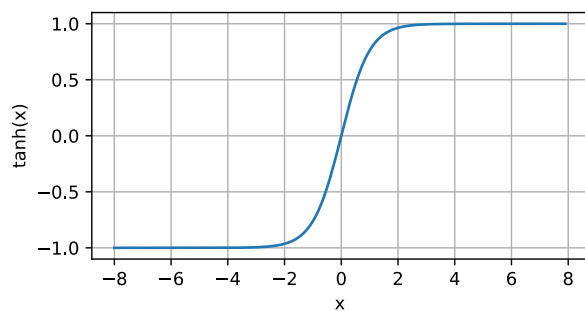



```
1 x.grad.data.zero_()
2 y.backward(torch.ones_like(x), retain_graph=True)
3 d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

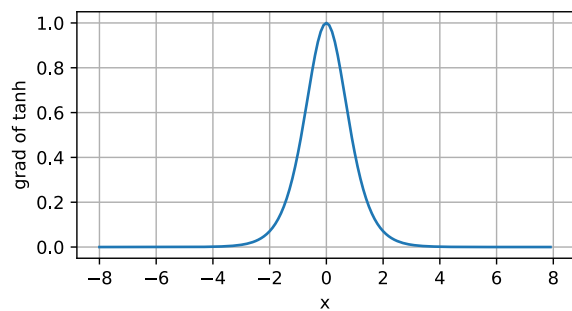


5.1.2.3. Tanh Function

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



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



5.2. Implementation of Multilayer Perceptrons

```
1 import torch
2 from torch import nn
3 from d2l import torch as d2l
```

5.2.1. Implementation from Scratch

5.2.1.1. Initialising Model Parameters

```

1 class MLPScratch(d2l.Classifier):
2     def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
3         super().__init__()
4         self.save_hyperparameters()
5         self.W1=nn.Parameter(torch.randn(num_inputs, num_hiddens)*sigma)
6         self.b1=nn.Parameter(torch.zeros(num_hiddens))
7         self.W2=nn.Parameter(torch.randn(num_hiddens, num_outputs)*sigma)
8         self.b2=nn.Parameter(torch.zeros(num_outputs))

```

5.2.1.2. Model

```

1 def relu(X):
2     a=torch.zeros_like(X)
3     return torch.max(X, a)

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

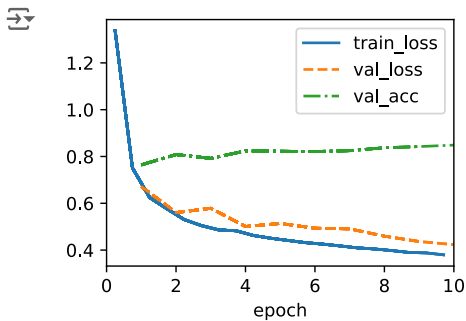
```

5.2.1.3. Training

```

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

```



5.2.2. Concise Implementation

5.2.2.1. Model

```

1 class MLP(d2l.Classifier):
2     def __init__(self, num_outputs, num_hiddens, lr):
3         super().__init__()
4         self.save_hyperparameters()
5         self.net=nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
6                                nn.ReLU(), nn.LazyLinear(num_outputs))

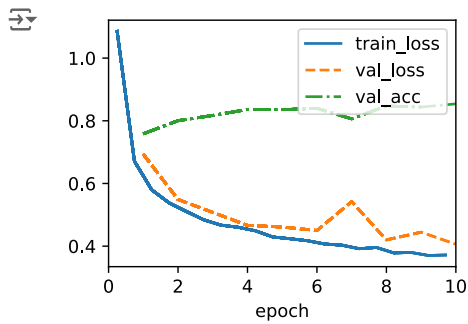
```

5.2.2. Training

```

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

```



5.3. Forward Propagation, Backward Propagation, and Computational Graphs

(No Codes)

Discussions and Exercises

Ch.2

Discussions / Exercises

2.1.

① Can we use '<' and '>' to compare each elements in tensor?

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

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

```
1 X<Y
```

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

```
1 X>Y
```

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

Answer: Yes.

② Can torch auto-calculate the result of sum between different dimensional tensor, especially for more than 2 dimension? (e.g. 3-dimensional tensor)

+) All dimensions have to be n times or 1/n times other dimensions to calculate. ($n \in \mathbb{N}$)

```
1 a=torch.arange(6).reshape((3, 1, 2))
2 b=torch.arange(8).reshape((1, 2, 4))
3 a, b
```

```
(tensor([[[[0, 1]],
          [[2, 3]],
          [[4, 5]]]],
        tensor([[[[0, 1, 2, 3],
          [4, 5, 6, 7]]]]))
```

```
1 a+b
```



```
RuntimeError                                Traceback (most recent call last)
<ipython-input-35-ca730b97bf8a> in <cell line: 1>()
----> 1 a+b
```

```
RuntimeError: The size of tensor a (2) must match the size of tensor b (4) at non-singleton dimension 2
```

```
1 a=torch.arange(6).reshape((3, 1, 2))
2 b=torch.arange(8).reshape((1, 4, 2))
3 a, b
4 a+b
```



```
tensor([[[[ 0,  2],
           [ 2,  4],
           [ 4,  6],
           [ 6,  8]],
         [[ 2,  4],
           [ 4,  6],
           [ 6,  8],
           [ 8, 10]],
         [[ 4,  6],
           [ 6,  8],
           [ 8, 10],
           [10, 12]]]])
```

```
1 a=torch.arange(12).reshape((3, 4, 1))
2 b=torch.arange(8).reshape((1, 4, 2))
3 a, b
4 a+b
```



```
tensor([[[[ 0,  1],
           [ 3,  4],
           [ 6,  7],
           [ 9, 10]],
         [[ 4,  5],
           [ 7,  8],
           [10, 11],
           [13, 14]],
         [[ 8,  9],
           [11, 12],
           [14, 15],
           [17, 18]]]])
```

Answer: At most 2 dimensions can be different, and others have to be the same.

✓ Ch.3

✓ Discussions / Exercises

✓ 3.1.

The key technique for optimising nearly every deep learning model consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called **gradient descent**.

The most naive application of gradient descent consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset. In practice, this can be **extremely slow**: we must pass over the entire dataset before making a single update, even if the update steps might be very powerful (Liu and Nocedal, 1989).

The other extreme is to consider **only a single example** at a time and to take update steps based on one observation at a time. The resulting algorithm, **stochastic gradient descent (SGD)** can be an effective strategy (Bottou, 2010), even for large datasets.

Unfortunately, SGD has drawbacks, both computational and statistical. One problem is that it can **take a lot longer** to process one sample at a time compared to a full batch.

A second problem is that some of the layers, such as batch normalisation (to be described in Section 8.5), **only work well when we have access to more than one observation at a time**.

The solution to both problems is to pick an intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a **minibatch** of observations (Li et al., 2014). This leads us to **minibatch stochastic gradient descent**.

✓ Ch.4

✓ Discussions / Exercises

✓ 4.4.

<Redifining Softmax Algorithm>

As computer processes float in a limited way, the traditional softmax are not useful when dealing with large or small value.

So, as

$$e^x \times e^y = e^{x+y}$$

we can change the variable from

$$\text{softmax}(X)_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}$$

to

$$\text{redifined_softmax}(X)_{ij} = \frac{\exp(X_{ij} - tI)}{\sum_k \exp(X_{ik} - tI)}$$

where t is the minimum of X.

✓ Ch.5

✓ Discussions / Exercises

✓ 5.1.

<Designing new activation function>

Looking the derivative of sigmoid function($\sigma(x)$), it is bell-shaped function.

Similarly, we can consider the Normal distribution, or T-distribution.

The c.d.f.(cumulative distribution function) of these functions might be a good activation function.

I will use normal distribution this time, as it was mentioned in Ch.3.

cf. Regarding T distribution,

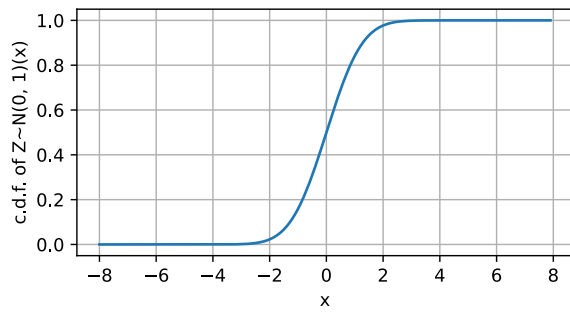
$$T = \frac{Z}{\sqrt{W/\nu}} \sim t(\nu)$$

where

$$Z \sim N(0, 1), W \sim \chi^2(\nu)$$

```
1 %matplotlib inline
2 import torch
3 from d2l import torch as d2l

1 # Function itself
2
3 x=torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
4 y=torch.special.ndtr(x)
5 d2l.plot(x.detach(), y.detach(), 'x', 'c.d.f. of Z~N(0, 1)(x)', figsize=(5, 2.5))
```



```
1 # Gradient
2
3 y.backward(torch.ones_like(x), retain_graph=True)
4 d2l.plot(x.detach(), x.grad, 'x', 'grad of c.d.f. of Z~N(0, 1)(x) (=p.d.f. of Z~N(0, 1))', figsize=(5, 2.5))
```

