COSE474 2024-2 HW 1

2023320119 김동현

2.1 Data Manipulation

In [1]:
```
import torch
```

In [2]:
```
x = torch.arange(12, dtype=torch.float32)
x
```

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

In [3]:
```
x.numel()
```

Out[3]:
```
12
```

In [4]:
```
x.shape
```

Out[4]:
```
torch.Size([12])
```

In [5]:
```
X = x.reshape(3, 4)
X
```

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

In [6]:
```
X = x.reshape(-1, 4)
X
```

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

In [7]:
```
torch.zeros((2, 3, 4))
```

Out[7]:
```
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.]]])
```

In [8]:
```
torch.ones((2, 3, 4))
```

Out[8]:
```
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.]]])
```

In [9]:
```
torch.randn(3, 4)
```

Out[9]:
```
tensor([[-0.3005, -1.0123, -0.8099,  0.8853],
        [ 1.2701, -1.0570, -0.7185,  1.3159],
        [ 1.3195,  0.9399,  0.8382, -2.5446]])
```

In [10]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

Out[10]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

In [11]:
```
X[-1], X[1:3]
```

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

In [12]:
```
X[1, 2] = 17
X
```

Out[12]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

In [13]:
```
X[:2, :] = 12
X
```

Out[13]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

In [14]:
```
torch.exp(x)
```

Out[14]:
```
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])
```

In [15]:
```
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
```

Out[15]:
```
(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.]))
```

In [16]:
```
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)
```

Out[16]:
```
(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.]]))
```

In [17]:
```
X == Y
```

Out[17]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

In [18]:
```
X.sum()
```

Out[18]:
```
tensor(66.)
```

In [19]:
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

Out[19]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

In [20]:
```
a + b
```

Out[20]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

In [21]:
```
before = id(Y)
Y = Y + X
id(Y) == before
```

Out[21]:
```
False
```

In [22]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 2311154767568
id(Z): 2311154767568
```

In [23]:
```
before = id(X)
X += Y
id(X) == before
```

Out[23]:
```
True
```

In [24]:
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

Out[24]: (numpy.ndarray, torch.Tensor)

In [25]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[25]: (tensor([3.5000]), 3.5, 3.5, 3)

2.2 Data Preprocessing

In [26]:
```
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"""
    )
```

In [27]:
```
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
```

In [28]:
```
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
```

In [29]:
```
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
```

In [30]:
```
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

Out[30]:
```
(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

In [31]:
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

Out[31]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

In [32]:
```
x = torch.arange(3)
x
```

Out[32]:
```
tensor([0, 1, 2])
```

In [33]:
```
x[2]
```

Out[33]:
```
tensor(2)
```

In [34]:
```
len(x)
```

Out[34]:
```
3
```

In [35]:
```
x.shape
```

Out[35]:
```
torch.Size([3])
```

In [36]:
```
A = torch.arange(6).reshape(3, 2)
A
```

Out[36]:
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

In [37]:
```
A.T
```

Out[37]:
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

In [38]:
```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

Out[38]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

In [39]:
```
torch.arange(24).reshape(2, 3, 4)
```

Out[39]:
```
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]]])
```

In [40]:
```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()
A, A + B
```

Out[40]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

In [41]:
```
A * B
```

Out[41]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

In [42]:
```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Out[42]:
```
(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]))
```

In [43]:
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

Out[43]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

In [44]:
```
A.shape, A.sum()
```

Out[44]:
```
(torch.Size([2, 3]), tensor(15.))
```

In [45]:
```
A.shape, A.sum(axis=0).shape
```

Out[45]:
```
(torch.Size([2, 3]), torch.Size([3]))
```

In [46]:
```
A.shape, A.sum(axis=1).shape
```

Out[46]:
```
(torch.Size([2, 3]), torch.Size([2]))
```

In [47]:
```
A.sum(axis = [0, 1]) == A.sum()
```

Out[47]:
```
tensor(True)
```

In [48]:
```
A.mean(), A.sum() / A.numel()
```

Out[48]:
```
(tensor(2.5000), tensor(2.5000))
```

In [49]:
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

Out[49]:
```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

In [50]:
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

Out[50]:
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

In [51]:
```
A / sum_A
```

Out[51]:
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

In [52]:
```
A.cumsum(axis=0)
```

Out[52]:
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

In [53]:
```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

Out[53]:
```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

In [54]:
```
torch.sum(x * y)
```

Out[54]:
```
tensor(3.)
```

In [55]:
```
A.shape, x.shape, torch.mv(A, x), A@x
```

Out[55]:
```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

In [56]:
```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

Out[56]:
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

In [57]:
```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

Out[57]:
```
tensor(5.)
```

In [58]:
```
torch.abs(u).sum()
```

Out[58]:
```
tensor(7.)
```

In [59]:
```
torch.norm(torch.ones((4, 9)))
```

Out[59]:
```
tensor(6.)
```

2.5 Automatic Differentitation

In [60]:
```
x = torch.arange(4.0)
x
```

Out[60]:
```
tensor([0., 1., 2., 3.])
```

In [61]:
```
x.requires_grad_(True)
x.grad
```

In [62]:
```
y = 2 * torch.dot(x, x)
y
```

Out[62]:
```
tensor(28., grad_fn=<MulBackward0>)
```

In [63]:
```
y.backward()
x.grad
```

Out[63]:
```
tensor([ 0.,  4.,  8., 12.])
```

In [64]:
```
x.grad == 4 * x
```

Out[64]:
```
tensor([True, True, True, True])
```

In [65]:
```
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

Out[65]:
```
tensor([1., 1., 1., 1.])
```

In [66]:
```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))
x.grad
```

Out[66]:
```
tensor([0., 2., 4., 6.])
```

In [67]:
```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

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

Out[67]:
```
tensor([True, True, True, True])
```

In [68]:
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

Out[68]:
```
tensor([True, True, True, True])
```

In [69]:
```
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
```

In [70]:
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In [71]:
```
a.grad == d / a
```

Out[71]:
```
tensor(True)
```

3.1 Linear Regression

In [72]:
```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

In [73]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [74]:
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

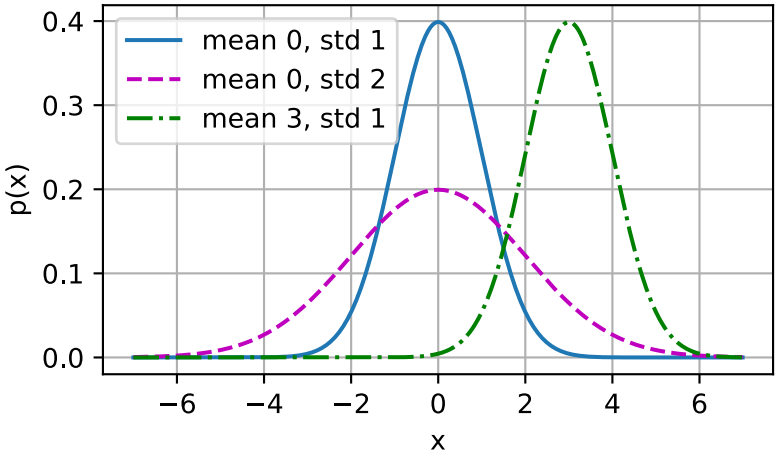
Out[74]:
```
'0.13664 sec'
```

In [75]:
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[75]:
```
'0.00100 sec'
```

In [76]:
```
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)
```

In [77]:
```
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

In [78]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

In [79]:
```
def add_to_class(Class):
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

In [80]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In [81]:
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

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

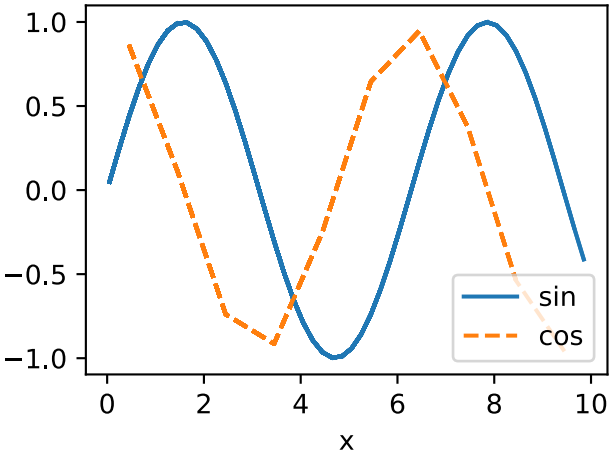
In [82]:
```
class HyperParameters:
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

In [83]:
```
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
```

In [84]:
```
class ProgressBoard(d2l.HyperParameters):
    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
```

In [85]:
```
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)
```



In [86]:
```
class Module(nn.Module, d2l.HyperParameters):
    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):
        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
```

In [87]:
```
class DataModule(d2l.HyperParameters):
    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)
```

In [88]:
```
class Trainer(d2l.HyperParameters):
    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

In [89]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

In [90]:
```
class LinearRegressionScratch(d2l.Module):
    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)
```

In [91]:
```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

In [92]:
```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

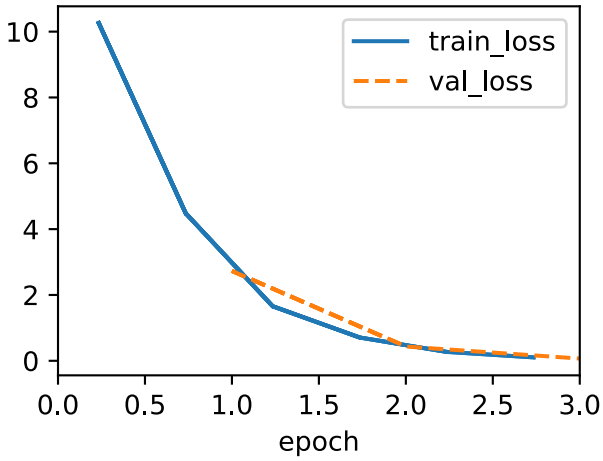
In [93]:
```
class SGD(d2l.HyperParameters):
    def __init__(self, params, lr):
        self.save_hyperparameters()

    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_()
```

In [94]:
```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

In [95]:
```
@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:
                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
```

In [96]:
```
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)
```



In [97]:
```
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([ 0.0801, -0.2437])
error in estimating b: tensor([0.2549])
```

4.2 The Image Classification Dataset

In [98]:
```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

In [99]:
```
class FashionMNIST(d2l.DataModule):
    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)
```

In [100…
```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

Out[100…
```
(60000, 10000)
```

In [101…
```
data.train[0][0].shape
```

Out[101…
```
torch.Size([1, 32, 32])
```

In [102…
```
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

In [103…
```
@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)
```

In [104…
```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

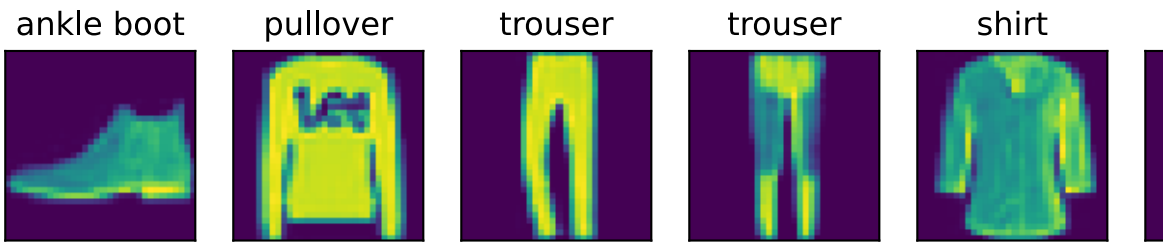
```
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

In [105…
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

Out[105…
```
'8.24 sec'
```

In [106…
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    raise NotImplementedError
```

In [107…
```
@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

In [108…
```
import torch
from d2l import torch as d2l
```

In [109…
```
class Classifier(d2l.Module):
    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)
```

In [110…
```
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

In [111…
```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    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

In [112…
```
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)
```

Out[112…
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

In [113…
```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition
```

In [114…
```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

Out[114…
```
(tensor([[0.2781, 0.1881, 0.1581, 0.1540, 0.2217],
         [0.2486, 0.1305, 0.1710, 0.1702, 0.2797]]),
 tensor([1.0000, 1.0000]))
```

In [115…
```
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]
```

In [116…
```
@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)
```

In [117…
```
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]
```

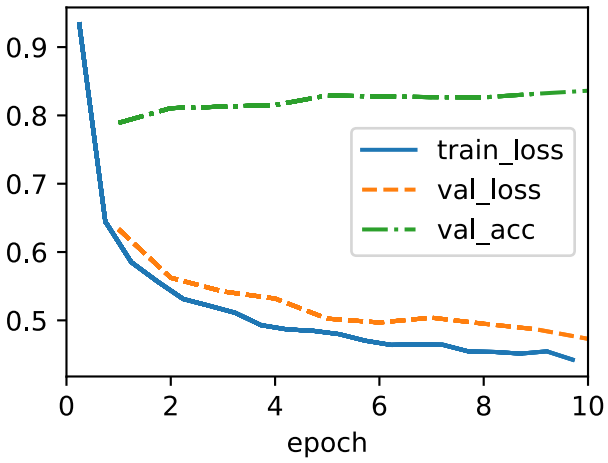
Out[117… tensor([0.1000, 0.5000])

In [118…
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[118… tensor(1.4979)

In [119…
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

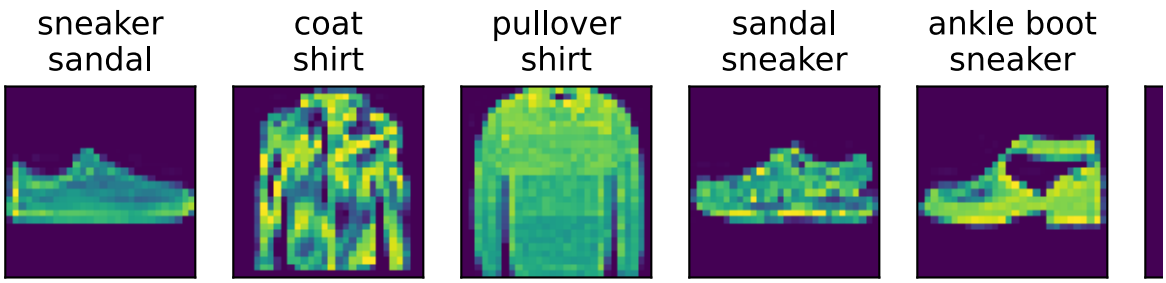
In [120…
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



In [121…
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[121… torch.Size([256])

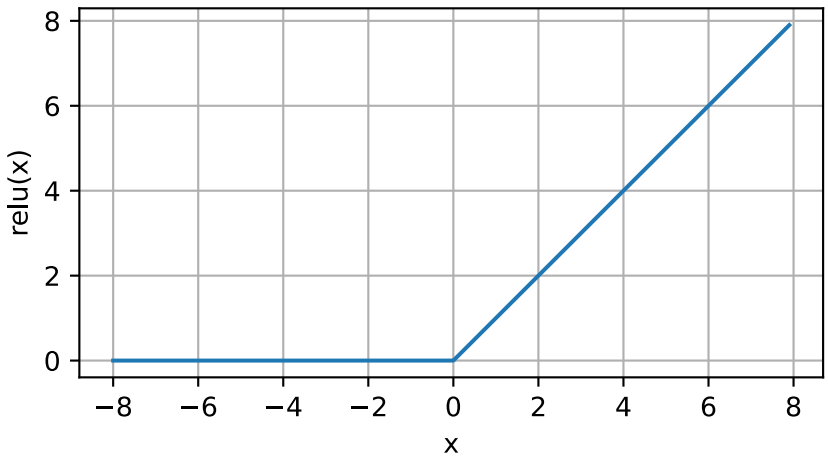
In [122…
```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



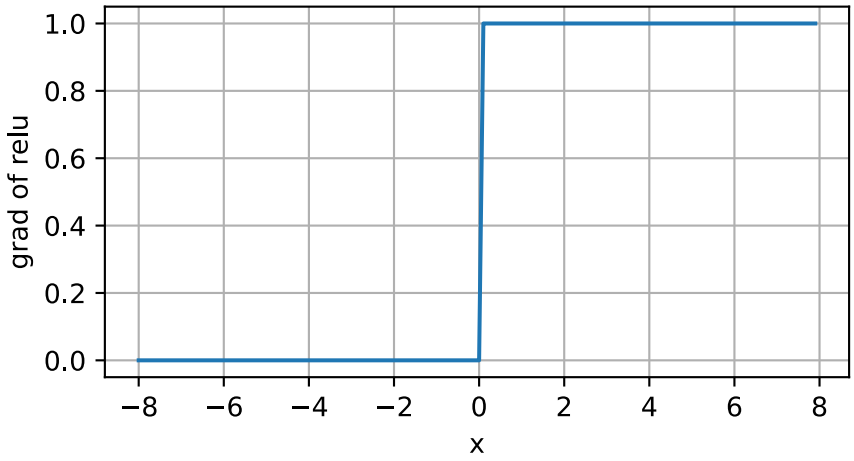
5.1 Multilayer Perceptrons

In [123…
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

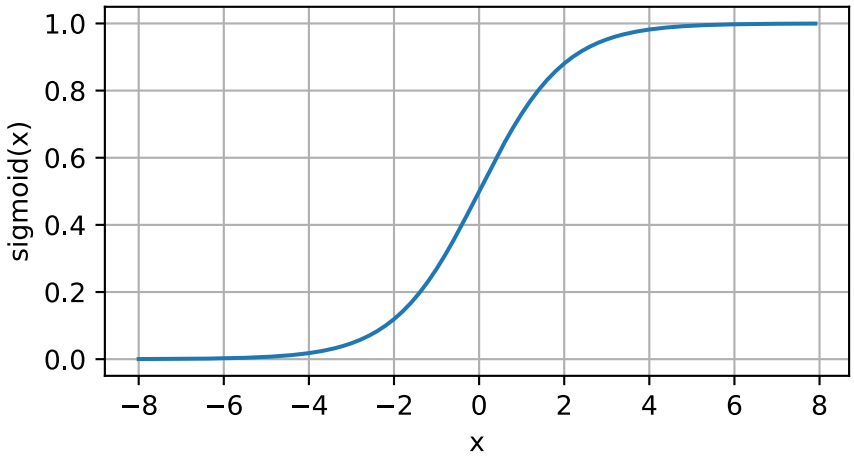
In [124…
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



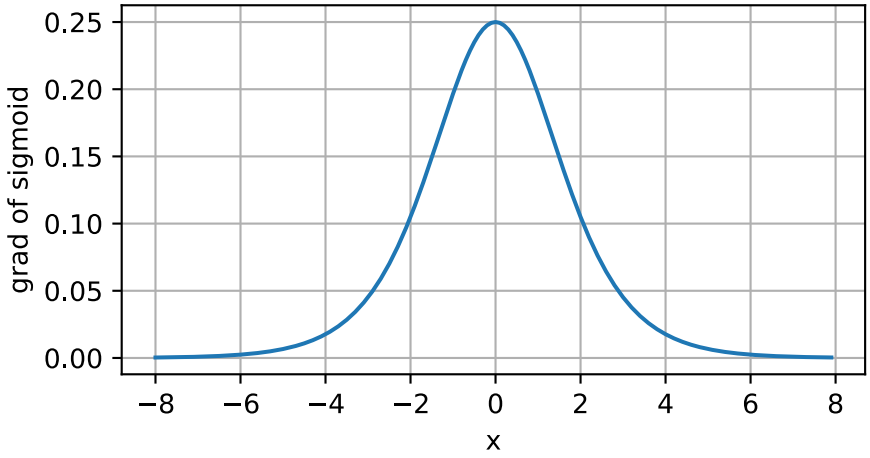
In [125…
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



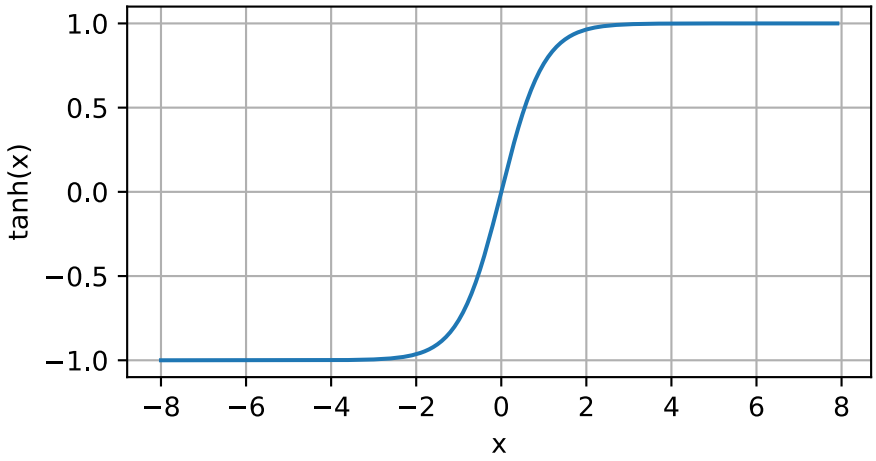
In [126…
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



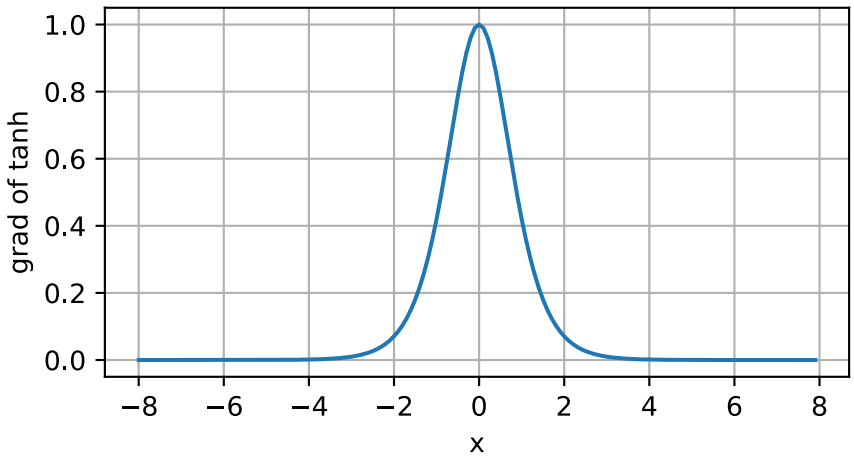
In [127…
```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



In [128…
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



In [129…
```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



5.2 Implementation of Multilayer Perceptrons

In [130…
```
import torch
from torch import nn
from d2l import torch as d2l
```

Note that for every layer, we must keep track of one weight matrix and one bias vector.
We use nn.Parameter to automatically register a class attribute as a parameter to be tracked by autograd.

In [131…
```
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))
```

We will implement the ReLU activation ourselves rather than invoking the built-in relu function directly.

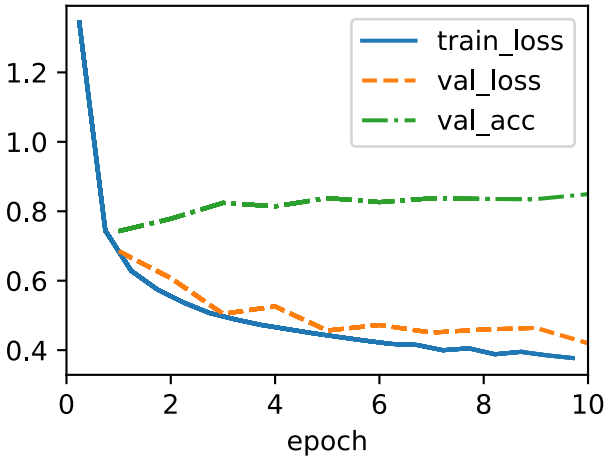
In [132…
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

Since we are disregarding spatial structure, we reshape each two-dimensional image into a flat vector of length num_inputs.

In [133…
```
@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
```

The training loop for MLPs is exactly the same as for softmax regression.

In [134…
```
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)
```

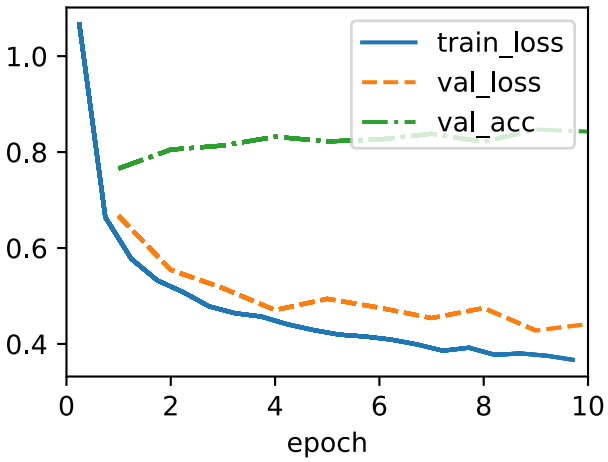


- Concise Implementation
  Compared with our concise implementation of softmax regression implementation, the only difference is that we add two fully connected layers where we previously added only one.
  The first is the hidden layer, the second is the output layer.

In [135…
```
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))
```

The training loop for Concise Implementation of MLPs is exactly the same as for softmax regression.

In [136…
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



Discussion

2.1 Data Manipulation

In the one-dimensional case, one axis needed for the data, a tensor is called vector. With two axes, a tensor is called a matrix.

Specifying every shape component to reshape is redundant.
because we already know our tensor's size, we can work out one component of the shape given the rest.

In [137…
```
x = torch.arange(12, dtype=torch.float32)
X = x.reshape(-1, 4)
X
```

Out[137…
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

In concatenating multiple tensors, tell the system along which axis to concatenate.
axis = 0 -> concatenate along rows / axis = 1 -> concatenate along columns

In [138…
```
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)
```

Out[138…
```
(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.]]))
```

Under certain conditions, even when shapes differ, we can still perform elementwise binary operations by invoking the broadcasting mechanism.

In [139…
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b, a + b
```

Out[139…
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]),
 tensor([[0, 1],
         [1, 2],
         [2, 3]]))
```

This(Python's id function issue) might be undesirable for two reasons.
First, we do not want to run around allocating memory unnecessarily all the time.
Second, we might point at the same parameters from multiple variables.

Fortunately, performing in-place operations is easy.
We can assign the result of an operation to a previously allocated array Y by using slice notation: Y[:]

2.2 Data Processing

In supervised learning, we train models to predict a designated target value, given some set of input values.

Missing values are called "bed bugs" of data science and these might be handled either via imputation or deletion.
Imputation replaces missing values with estimates of their values while deletion simply discards either those rows or those columns that contain missing values.

All the entries in inputs and targets are numerical, we can load them into a tensor.

2.3 Linear Algebra

We denote scalars by ordinary lower-cased letters (e.g. x, y, z) and the space of all (continuous) real-valued scalars by $\mathbb{R}$.
The symbol $\in$ denotes membership in a set.
Scalars are implemented as tensors that contain only one element.

As with their code counterparts, we call these scalars the elements of the vector.
When vectors represent examples from real-world datasets, their values hold some real-world significance.
We denote vectors by bold lowercase letters, (e.g., $\mathbf{x}, \mathbf{y}, \mathbf{z}$).
Vectors are implemented as 1st-order tensors.

To indicate that a vector contains n elements, we write $x \in \mathbb{R}^n$.
Formally, we call n the dimensionality of the vector.

The shape is a tuple that indicates a tensor's length along each axis.

we use order to refer to the number of axes and dimensionality exclusively to refer to the number of components.

We denote matrices by bold capital letters (e.g.,$\mathbf{X}, \mathbf{Y}, \mathbf{Z}$), and represent them in code by tensors with two axes.
The expression $\mathbf{A} \in \mathbb{R}^{m*n}$ indicates that a matrix $\mathbf{A}$ contains $m * n$ real-valued scalars, arranged as m rows and n columns.
When m = n, we say that a matrix is square.

We can convert any appropriately sized $m * n$ tensor into an $m * n$ matrix by passing the desired shape to reshape.

When we exchange a matrix's rows and columns, the result is called its transpose.
Formally, we signify a matrix $\mathbf{A}$'s transpose by $\mathbf{A}^\top$ and if $\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} == a_{ji}$ for all i and j.

Matrices are useful for representing datasets.
Typically, rows correspond to individual records and columns correspond to distinct attributes.

Tensors give us a generic way of describing extensions to nth-order arrays.
We call software objects of the tensor class "tensors" precisely because they too can have arbitrary numbers of axes.
Tensors will become more important when we start working with images.
Each image arrives as a 3rd-order tensor with axes corresponding to the height, width, and channel.
Furthermore, a collection of images is represented in code by a 4th-order tensor, where distinct images are indexed along the first axis.

The elementwise product of two matrices is called their Hadamard product (denoted ⊙)

Invoking the sum function reduces a tensor along all of its axes, eventually producing a scalar.
Our libraries also allow us to specify the axes along which the tensor should be reduced.
To sum over all elements along the rows (axis 0), we specify axis=0 in sum.

In [140…
```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
A.shape, A.sum(axis=0).shape
```

Out[140…
```
(torch.Size([2, 3]), torch.Size([3]))
```

A related quantity is the mean, also called the average.
We calculate the mean by dividing the sum by the total number of elements.

It can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean.
This matters when we want to use the broadcast mechanism.

In [141…
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

Out[141…
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

For instance, since sum_A keeps its two axes after summing each row, we can divide A by sum_A with broadcasting to create a matrix where each row sums up to 1.

In [142…
```
A / sum_A
```

Out[142…
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

If we want to calculate the cumulative sum of elements of A along some axis, say axis=0 (row by row), we can call the cumsum function.

In [143…
```
A.cumsum(axis=0)
```

Out[143…
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their dot product $\mathbf{x}^\top \mathbf{y}$ (also known as inner product, <x,y>) is a sum over the products of the elements at the same position.

To express a matrix–vector product in code, we use the mv function.
Python has a convenience operator @ that can execute both matrix–vector and matrix–matrix products (depending on its arguments).
Thus we can write A@x.

In [144…
```
x = torch.arange(3, dtype=torch.float32)
A.shape, x.shape, torch.mv(A, x), A@x
```

Out[144…
```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

The norm of a vector tells us how big it is.
A norm is a function $\|\cdot\|$ that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector $\mathbf{x}$, if we scale (all elements of) the vector by a scalar $\alpha \in \mathbb{R}$, its norm scales accordingly:
   $\|\alpha\mathbf{x}\| = \|\alpha\|\|\mathbf{x}\|$
2. For any vectors $\mathbf{x}$ and $\mathbf{y}$: norms satisfy the triangle inequality:
   $\|\mathbf{x} + \mathbf{y}\| \leq |\mathbf{x}\| + |\mathbf{y}\|$
3. The norm of a vector is nonnegative and it only vanishes if the vector is zero:
   $|\mathbf{x}\| > 0$ for all $|\mathbf{x}\| \neq 0$

Many functions are valid norms and different norms encode different notions of size.
Euclidean norm is called the l_2 norm and expressed as
$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$

Both the l_2 and l_1 norms are special cases of the more general l_p norms:
$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$

Frobenius norm is much easier to compute and defined as the square root of the sum of the squares of a matrix's elements:
$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}$

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.
To calculate derivatives, automatic differentiation works backwards through this graph applying the chain rule.
The computational algorithm for applying the chain rule in this fashion is called backpropagation.

Differentiating the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to the column vector $\mathbf{x}$.

Before we calculate the gradient of y with respect to x, we need a place to store it.
In general, we avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory.

In [145…
```
x.requires_grad_(True)
x.grad
```

In [146…
```
y = 2 * torch.dot(x, x)
y
```

Out[146…
```
tensor(10., grad_fn=<MulBackward0>)
```

We can now take the gradient of y with respect to x by calling its backward method.

In [147…
```
y.backward()
x.grad
```

Out[147…
```
tensor([0., 4., 8.])
```

When y is a vector, the most natural representation of the derivative of y with respect to a vector x is a matrix called the Jacobian that contains the partial derivatives of each component of y with respect to each component of x.
Invoking backward on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar.

We use the input to create some auxiliary intermediate terms for which we do not want to compute a gradient.
In this case, we need to detach the respective computational graph from the final result.

In [148…
```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

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

Out[148…
```
tensor([True, True, True])
```

3.1 Linear Regression

Regression problems pop up whenever we want to predict a numerical value.

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

Linear regression flows from a few simple assumptions.
First, we assume that the relationship between features $\mathbf{x}$ and target y is approximately linear, i.e., that the conditional mean $E[Y \mid X = \mathbf{x}]$ can be expressed as a weighted sum of the features.
Next, we can impose the assumption that any such noise is well behaved, following a Gaussian distribution.

- Model
  At the heart of every solution is a model that describes how features can be transformed into an estimate of the target.
  The weights determine the influence of each feature on our prediction.
  The bias determines the value of the estimate when all features are zero.
  Given a dataset, our goal is to choose the weights w and the bias b that, on average, make our model's predictions fit the true prices observed in the data as closely as possible.
  Collecting all features into a vector $\mathbf{x} \in \mathrm{r}^d$ and all weights into a vector $\mathbf{w} \in \mathrm{r}^d$, we can express our model compactly via the dot product between $\mathbf{w}$ and $\mathbf{x}$:
  $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$

- Loss Function
  Loss functions quantify the distance between the real and predicted values of the target.
  The loss will usually be a nonnegative number where smaller values are better and perfect predictions incur a loss of 0.
  To measure the quality of a model on the entire dataset of n examples, we simply average (or equivalently, sum) the losses on the training set:
  $L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$
  When training the model, we seek parameters (w^*, b^*) that minimize the total loss across all training examples:
  $\mathbf{w}^*, b^* = \arg\min_{\mathbf{w}, b} L(\mathbf{w}, b)$

- Analytic Solution
  Linear regression presents us with a surprisingly easy optimization problem. First, we can subsume the bias b into the parameter w by appending a column to the design matrix consisting of all 1s.
  Taking the derivative of the loss with respect to w and setting it equal to zero yields.

- Minibatch Stochastic Gradient Descent
  Fortunately, even in cases where we cannot solve the models analytically, we can still often train models effectively in practice.
  The key technique for optimizing nearly every deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function.
  This algorithm is called gradient descent.
  Picking an intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a minibatch of observations.

- Predications
  Given the model $\mathbf{w}^\top \mathbf{x} + b$, we can now make predictions for a new example.

When training our models, we typically want to process whole minibatches of examples simultaneously.
Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.

In [149…
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

First, we add them, one coordinate at a time, using a for-loop.

In [150…
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

Out[150… `'0.20961 sec'`

Alternatively, we rely on the reloaded + operator to compute the elementwise sum.

In [151…
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[151… `'0.00000 sec'`

The second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups.

It turns out that the normal distribution and linear regression with squared loss share a deeper connection than common parentage.

While linear models are not sufficiently rich to express the many complicated networks that we will introduce in this book, (artificial) neural networks are rich enough to subsume linear models as networks in which every feature is represented by an input neuron, all of which are connected directly to the output.
The diagram highlights the connectivity pattern, such as how each input is connected to the output, but not the specific values taken by the weights or biases.
In summary, we can think of linear regression as a single-layer fully connected neural network.
Certainly, the high-level idea that many such units could be combined, provided they have the correct connectivity and learning algorithm, to produce far more interesting and complex behavior than any one neuron alone could express arises from our study of real biological neural systems.

3.2 Object-Oriented Design for Implementation

Linear regression is one of the simplest machine learning models.
Training it, however, uses many of the same components that other models in this book require.
Therefore, it is worth designing some of the APIs that we use throughout.

The first utility function allows us to register functions as methods in a class after the class has been created.

The second one is a utility class that saves all arguments in a class's `__init__` method as class attributes.

The final utility allows us to plot experiment progress interactively while it is going on.

The Module class is the base class of all models we will implement.
The first, `__init__`, stores the learnable parameters, the training_step method accepts a data batch to return the loss value, and finally, configure_optimizers returns the optimization method, or a list of them, that is used to update the learnable parameters.
Optionally we can define validation_step to report the evaluation measures.

In [152…

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

The DataModule class is the base class for data.
The `__init__` method is used to prepare the data. This includes downloading and preprocessing if needed.
The train_dataloader returns the data loader for the training dataset.
There is an optional val_dataloader to return the validation dataset loader.

In [153…

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

The Trainer class trains the learnable parameters in the Module class with data specified in DataModule.
`fit` method iterates over the entire dataset max_epochs times to train the model.

In [154…
```
class Module(nn.Module, d2l.HyperParameters):
    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):
        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
```

3.4 Linear Regression Implementation from Scratch

To illustrate how to train more general neural networks, we teach you how to use minibatch SGD. At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of the loss with respect to the parameters.
Next, we update the parameters in the direction that may reduce the loss.

In [155…
```
class SGD(d2l.HyperParameters):
    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_()
```

In [156…
```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

In each epoch, we iterate through the entire training dataset, passing once through every example.
In each iteration, we grab a minibatch of training examples, and compute its loss through the model's training_step method.
Then we compute the gradients with respect to each parameter.
Finally, we will call the optimization algorithm to update the model parameters.

Before training the model, We need some training data.
Using the SyntheticRegressionData class and pass in some ground truth parameters.

we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop.

4.1 Softmax Regression

we have to choose how to represent the labels.
Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent {dog, cat, chicken} respectively.
In our case, a label y would be a three-dimensional vector, with (1, 0, 0) corresponding to "cat", (0, 1, 0) to "chicken", and (0, 0, 1) to "dog"
$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$

- Linear Model
  In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class.
  To address classification with linear models, we will need as many affine functions as we have outputs.
  Each output corresponds to its own affine function.
  In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights(w) and 3 scalars to represent the biases(b).

- The Softmax
  Assuming a suitable loss function, we could try, directly, to minimize the difference between o and the labels y.
  While it turns out that treating classification as a vector-valued regression

problem works surprisingly well, it is nonetheless unsatisfactory in the following ways:

There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.

There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

Both aspects render the estimation problem difficult to solve and the solution very brittle to outliers.

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

Softmax is way to accomplish this goal.

- Vecotrization
  To improve computational efficiency, we vectorize calculations in minibatches of data.
  This accelerates the dominant operation into a matrix–matrix product $\mathbf{XW}$.

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

- Entropy
  Entropy in information theory is to quantify the amount of information contained in data.

- Surprisal
  Easy to predict, easy to compress.
  However if we cannot perfectly predict every event, then we might sometimes be surprised.

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

In short, we can think of the cross-entropy classification objective in two ways:

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

Flow Summary

1. Softmax regression is used for multiclass classification problem.
2. For a given input, Calculate probability of which of the class the model belongs to and Choose class that has highest probability.

3. The difference between this probability distribution (Prediction probability distribution) and real label is calculated by Cross Entropy Loss

4.2 The Image Classification Dataset

Since the Fashion-MNIST dataset is so useful, all major frameworks provide preprocessed versions of it.

The images are grayscale and upscaled to 32 * 32 pixels in resolution above.
most modern image data has three channels (red, green, blue) and that hyperspectral images can have in excess of 100 channels (the HyMap sensor has 126 channels). By convention we store an image as a c * h * w tensor, where c is the number of color channels, h is the height and w is the width.

Recall that at each iteration, a data iterator reads a minibatch of data with size batch_size.
We also randomly shuffle the examples for the training data iterator.

In [157…
```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([256, 1, 28, 28]) torch.float32 torch.Size([256]) torch.int64
```

In [158…
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

Out[158…
```
'8.19 sec'
```

In general, it is a good idea to visualize and inspect data that you are training on.

4.3 The Base Classification Model

In [159…
```
class Classifier(d2l.Module):
    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)
```

In the validation_step we report both the loss value and the classification accuracy on a validation batch.
We draw an update for every num_val_batches batches. This has the benefit of generating the averaged loss and accuracy on the whole validation data.

We typically choose the class with the highest predicted probability whenever we must output a hard prediction.
When predictions are consistent with the label class y, they are correct.
The classification accuracy is the fraction of all predictions that are correct.

4.4 Softmax Regression Implementation from Scratch

The number of epochs (max_epochs), the minibatch size (batch_size), and learning rate (lr) are adjustable hyperparameters.
That means that while these values are not learned during our primary training loop, they still influence the performance of our model.

After training is complete, our model is ready to classify some images.

We are more interested in the images we label incorrectly.
We visualize them by comparing their actual labels (first line of text output) with the predictions from the model (second line of text output).

5.1 Multilayer Perceptrons

The simplest deep networks are called multilayer perceptrons, and they consist of multiple layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence).

Softmax regression, implementing the algorithm from scratch allowed us to train classifiers capable of recognizing 10 categories of clothing from low-resolution images.

Linearity implies the weaker assumption of monotonicity.
One way to handle this might be to postprocess our outcome such that linearity becomes more plausible, by using the logistic map.
but, it is less obvious that we could address the problem with a simple preprocessing fix.
We used observational data to jointly learn both a representation via hidden layers and a linear predictor that acts upon that representation.

We can overcome the limitations of linear models by incorporating one or more hidden layers.
The easiest way to do this is to stack many fully connected layers on top of one another.
This architecture is commonly called a multilayer perceptron, often abbreviated as MLP.

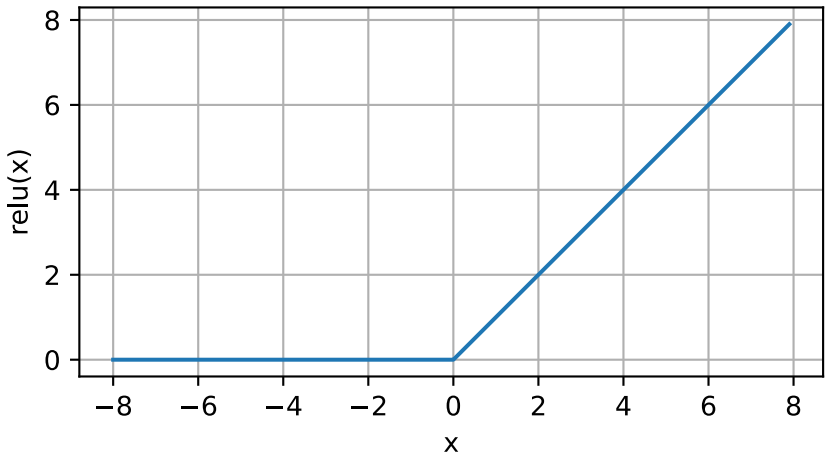
For a one-hidden-layer MLP whose hidden layer has h hidden units, we denote by $\mathbf{H} \in \mathbb{R}^{n*h}$ the outputs of the hidden layer, which are hidden representations.
After adding the hidden layer, our model now requires us to track and update additional sets of parameters.
In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear activation function σ to be applied to each hidden unit following the affine transformation.

In general, with activation functions in place, it is no longer possible to collapse our MLP into a linear model.

Activation functions decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it.

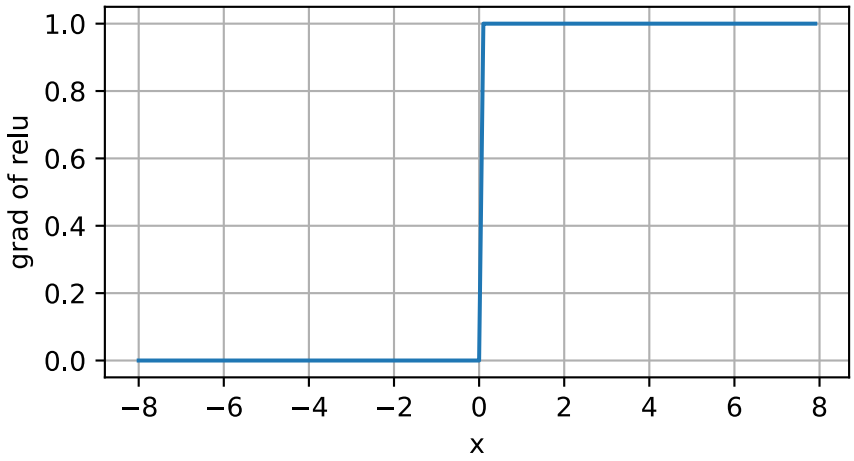
- ReLU Function
  The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the rectified linear unit (ReLU).
  $\mathrm{ReLU}(x) = \max(x, 0)$

In [160…
```
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))
```



When the input is negative, the derivative of the ReLU function is 0, and when the input is positive, the derivative of the ReLU function is 1.

In [161…
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



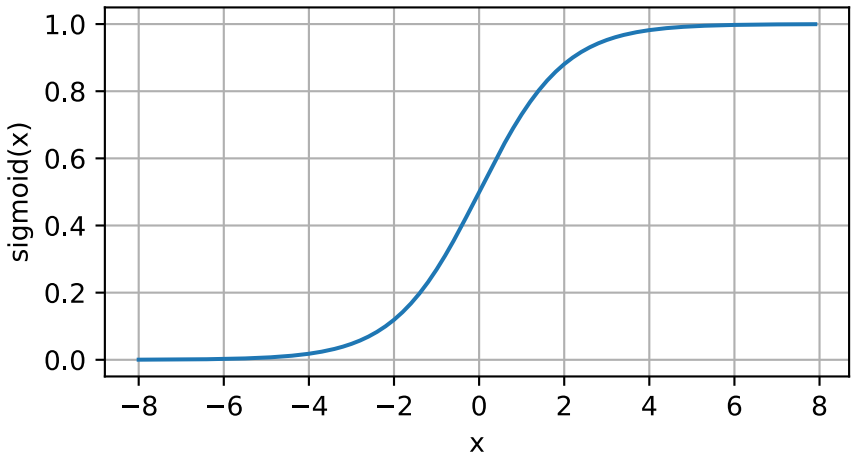
The reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through.
This makes optimization better behaved and it mitigated the well-documented

problem of vanishing gradients that plagued previous versions of neural networks (more on this later).

- Sigmoid Function
  The sigmoid is often called a squashing function: it squashes any input in the range -inf, inf to some value in the range (0, 1) $\text{sigmoid}(x) = \frac{1}{1+\exp(-x)}$

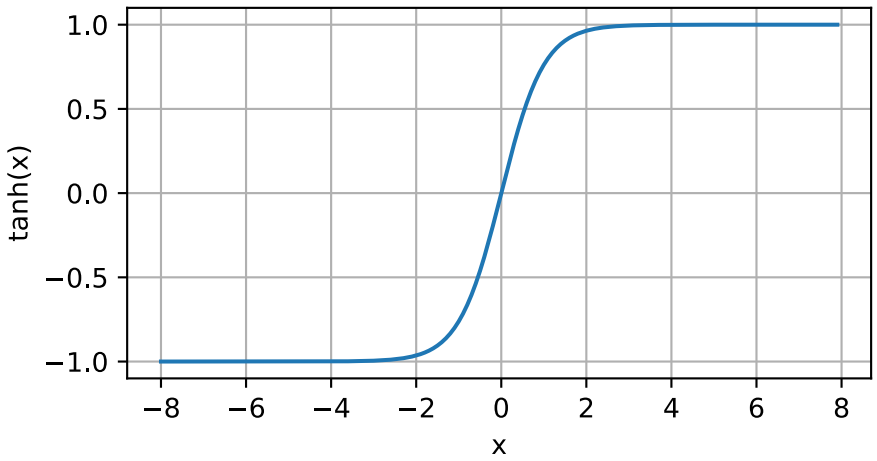
  Sigmoids are still widely used as activation functions on the output units when we want to interpret the outputs as probabilities for binary classification problems

In [162…
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



- Tanh Function
  Tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between -1 and 1
  $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$

In [163…
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



5.2 Implementation of Multilayer Perceptrons

Note that for every layer, we must keep track of one weight matrix and one bias vector.
We use nn.Parameter to automatically register a class attribute as a parameter to be tracked by autograd.

Since we are disregarding spatial structure, we reshape each two-dimensional image into a flat vector of length num_inputs.

The training loop for MLPs is exactly the same as for softmax regression.

- Concise Implementation
  Compared with our concise implementation of softmax regression implementation, the only difference is that we add two fully connected layers where we previously added only one.
  The first is the hidden layer, the second is the output layer.
  The training loop for Concise Implementation of MLPs is exactly the same as for softmax regression.

5.3. Forward Propagation, Backward Propagation, and Computational Graphs

Forward propagation (or forward pass) refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.
Backpropagation refers to the method of calculating the gradient of neural network parameters.
In short, backpropagation traverses the network in reverse order, from the output to the input layer, according to the chain rule from calculus.
When training neural networks, forward and backward propagation depend on each other.
In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path.
These are then used for backpropagation where the compute order on the graph is reversed.