Coding

2.1 Data Manipulation

2.1.1. Getting started

In [5]:
```python
import torch
```

In [6]:
```python
x = torch.arange(12, dtype=torch.float32) # 텐서 생성
x
```

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

In [7]:
```python
x.numel() ## number of element
```

Out[7]:
```
12
```

In [8]:
```python
x.shape ## size of x
```

Out[8]:
```
torch.Size([12])
```

In [9]:
```python
X = x.reshape(3,4) ## reshape tensor x from 1*12 to 3*4
X
```

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

In [10]:
```python
torch.zeros((2,3,4)) ## a 2*3*4 tensor filled with zeros
```

Out[10]:
```
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 [11]:
```
torch.ones((2,3,4)) ## a 2*3*4 tensor filled with ones
```

Out[11]:
```
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 [12]:
```
torch.randn(3,4) ## a 3*4 tensor filled with random numbers
```

Out[12]:
```
tensor([[-0.5003,  1.1148, -0.7721, -0.3903],
        [-1.8033,  0.1619,  1.5366,  0.1537],
        [ 1.0607, -2.2654,  0.0866, -0.9929]])
```

In [13]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

Out[13]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

2.1.2. Indexing and Slicing

In [14]:
```
X[-1], X[1:3]
```

Out[14]:
```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

In [15]:
```
X[1,2] = 17
```

```
X
```

Out[15]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

In [16]:
```
X[:2,:] = 12 ## fill elements whose indexes are [0~1, any] with 12
X
```

Out[16]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

2.1.3. Operations

In [17]:
```
torch.exp(x)
# e^12 * 8, e^8, e^9, e^10, e^11, e^12
# Due to reassignment of X, the values of x are also changed.
```

Out[17]:
```
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 [18]:
```
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[18]:
```
(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 [19]:
```
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)
# concatenate X and Y by choosing dimension between 0(vert) and 1(horz)
```

```
Out[19]: (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 [20]: X == Y # return a tensor where each element expresses True or False
```

```
Out[20]: tensor([[False,  True, False,  True],
                 [False, False, False, False],
                 [False, False, False, False]])
```

```
In [21]: X.sum()
```

```
Out[21]: tensor(66.)
```

2.1.4 Broadcasting

```
In [22]: a = torch.arange(3).reshape((3, 1))
         b = torch.arange(2).reshape((1, 2))
         a, b
```

```
Out[22]: (tensor([[0],
                  [1],
                  [2]]),
          tensor([[0, 1]]))
```

```
In [23]: a + b # adding a(3*1) with b(1*2) resulting to a 3*2 tensor
```

```
Out[23]: tensor([[0, 1],
                 [1, 2],
                 [2, 3]])
```

Saving Memory

In [24]:
```
before = id(Y)
Y = Y + X
id(Y) == before # the id are different!
```

Out[24]: False

In [25]:
```
Z = torch.zeros_like(Y)
print("id(Z):", id(Z))
Z[:] = X+Y # we can maintain memory location by using slice notation!
print("id(Z):", id(Z))
```

```
id(Z): 6381521808
id(Z): 6381521808
```

In [26]:
```
before = id(X)
X += Y # we can use += operator as well!
id(X) == before
```

Out[26]: True

2.1.6. Conversion to Other Python Objects

In [27]:
```
A = X.numpy() # we can convert a pytorch tensor to a numpy array!
B = torch.from_numpy(A) # we can convert a numpy array to a pytorch tensor!
type(A), type(B)
```

Out[27]: (numpy.ndarray, torch.Tensor)

In [28]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[28]: (tensor([3.5000]), 3.5, 3.5, 3)

2.2. Data Preprocessing

2.2.1. Reading the Dataset

In [29]:
```
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 [30]:
```
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms RoofType   Price
0       NaN      NaN  127500
1       2.0      NaN  106000
2       4.0    Slate  178100
3       NaN      NaN  140000
```

2.2.2. Data Preparation

In [31]:
```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True) # we have to convert all data to numerical form!
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 [32]:
```
inputs = inputs.fillna(inputs.mean()) # we can fill NaN fields with the mean value!
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 [33]:
```
import torch
# we converts pandas data to numpy array, and then converts again to pytorch tensors!
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

Out[33]:
```
(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

In [34]:
```
import torch
```

In [35]:
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

Out[35]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

2.3.2. Vectors

In [36]:
```
x = torch.arange(3)
x
```

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

In [37]:
```
print(x[2])
print(len(x))
print(x.shape)
```

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

2.3.3. Matrices

In [38]:
```
A = torch.arange(6).reshape(3, 2)
A
```

Out[38]:
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

In [39]:
```
A.T # Transpose of A
```

Out[39]:
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

In [40]:
```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T # A is symmetric
```

Out[40]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

2.3.4. Tensors

In [41]:
```
torch.arange(24).reshape(2, 3, 4) # A tensor with 3rd dimension
```

Out[41]:
```
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

In [42]:
```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()  # copy of A
A, A + B
```

Out[42]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

In [43]:
```
A * B # scalar multiplication
```

Out[43]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

In [44]:
```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Out[44]:
```
(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

In [45]:
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum() # sigma of X
```

Out[45]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

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

Out[46]:
```
(torch.Size([2, 3]), tensor(15.))
```

In [47]:
```
A.shape, A.sum(axis=0).shape # size of each row (along column)
```

Out[47]:
```
(torch.Size([2, 3]), torch.Size([3]))
```

In [48]:
```
A.shape, A.sum(axis=1).shape # size of each column (along row)
```

Out[48]:
```
(torch.Size([2, 3]), torch.Size([2]))
```

In [49]:
```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

Out[49]:
```
tensor(True)
```

In [50]:
```
A.mean(), A.sum() / A.numel()
```

Out[50]:
```
(tensor(2.5000), tensor(2.5000))
```

In [51]:
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0] # along column...
```

Out[51]:
```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

2.3.7. Non-Reduction Sum

In [52]:
```
sum_A = A.sum(axis=1, keepdims=True) # keep the size of rows!
sum_A, sum_A.shape
```

Out[52]:
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

In [53]:
```
A / sum_A
```

Out[53]:
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

In [54]:
```
A.cumsum(axis=0) # cumulative sum along column!
```

Out[54]:
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

2.3.8. Dot Products

In [55]:
```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

Out[55]:
```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

In [56]:
```
torch.sum(x * y) # dot product == elementwise multiplication
```

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

2.3.9. Matrix-Vector Products

In [57]:
```
A.shape, x.shape, torch.mv(A, x), A@x # get Ax
```

Out[57]:
```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

2.3.10. Matrix-Matrix Multiplication

In [58]:
```
B = torch.ones(3, 4)
torch.mm(A, B), A@B # torch.mm == @
```

Out[58]:
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

2.3.11. Norms

In [59]:
```
u = torch.tensor([3.0, -4.0])
torch.norm(u) # l2 norm (Euclidean norm)
```

Out[59]:
```
tensor(5.)
```

In [60]:
```
torch.abs(u).sum() # l1 norm (Manhattan distance)
```

Out[60]:
```
tensor(7.)
```

In [61]:
```
torch.norm(torch.ones((4, 9))) # Frobenius norm
```

Out[61]:
```
tensor(6.)
```

2.5. Automatic Differentiation

In [62]:
```
import torch
```

In [63]:
```
x = torch.arange(4.0)
x
```

Out[63]:
```
tensor([0., 1., 2., 3.])
```

In [64]:
```
x.requires_grad_(True) # activate gradient of x
x.grad # Not yet!
```

In [65]:
```
y = 2 * torch.dot(x, x) # y = 2((x^T)x)
y # y is a scalar function of vector x
```

Out[65]:
```
tensor(28., grad_fn=<MulBackward0>)
```

In [66]:
```
y.backward()
x.grad
```

Out[66]: tensor([0., 4., 8., 12.])

In [67]:
```
x.grad == 4*x # grad(y) = 4x
```

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

In [68]:
```
x.grad.zero_()
y = x.sum() # grad(y) = [1,1,1,1]
y.backward()
x.grad
```

Out[68]: tensor([1., 1., 1., 1.])

2.5.2. Backward for Non-Scalar Variables

In [69]:
```
x.grad.zero_()
y = x*x # grad(y) = 2x
y.backward(gradient=torch.ones(len(y))) # ?
x.grad
```

Out[69]: tensor([0., 2., 4., 6.])

2.5.3. Detaching Computation

In [70]:
```
x.grad.zero_()
y = x * x
u = y.detach() # new variable u whose value is equal to y, but is detached from x!
z = u * x

z.sum().backward()
x.grad == u # grad(z) != 3*x*x, rather grad(z) == u
```

Out[70]: tensor([True, True, True, True])

In [71]:
```
x.grad.zero_()
y.sum().backward()
```

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

Out[71]:

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

2.5.4. Gradients and Python Control Flow

In [72]:

```
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 [73]:

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In [74]:

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

Out[74]:

```
tensor(True)
```

3.1. Linear Regression

In [75]:

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

3.1.2. Vectorization for Speed

In [76]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [77]:
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i] # Slow!
f'{time.time() - t:.5f} sec'
```

Out[77]:
```
'0.03111 sec'
```

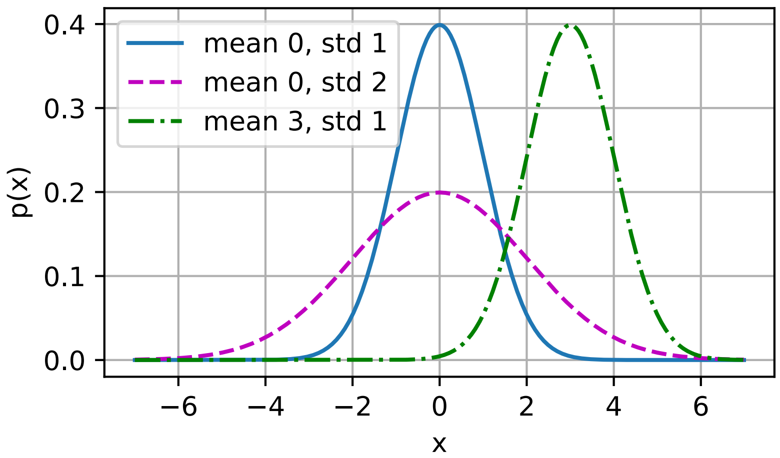
In [78]:
```
t = time.time()
d = a + b # fast!!!
f'{time.time() - t:.5f} sec'
```

Out[78]:
```
'0.00030 sec'
```

3.1.3. The Normal Distribution and Squared Loss

In [79]:
```
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 [80]:
```
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 [81]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

3.2.1. Utilities

In [82]:
```
def add_to_class(Class): #@save # register functions to class!
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

In [83]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In [84]:
```python
@add_to_class(A)
def do(self): # add do() to class A
    print('Class attribute "b" is', self.b)

a.do()
```

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

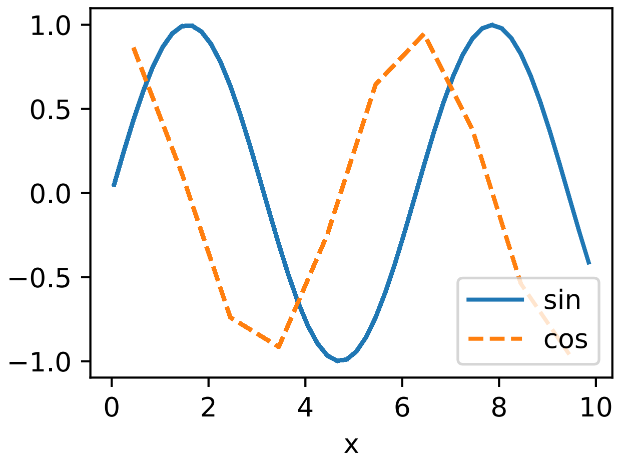
In [85]:
```python
class HyperParameters: #@save # save all args in __init__ functions to class member variables!
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

In [86]:
```python
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 [87]:
```python
class ProgressBoard(d2l.HyperParameters):  #@save
    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 [88]:
```python
board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)
```



3.2.2. Models

In [89]:
```
class Module(nn.Module, d2l.HyperParameters):  #@save
    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
```

3.2.3. Data

In [90]:

```
class DataModule(d2l.HyperParameters):  #@save
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

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

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

3.2.4. Training

In [91]:

```
class Trainer(d2l.HyperParameters):  #@save
    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 [92]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

In [93]:
```
class LinearRegressionScratch(d2l.Module):  #@save
    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 [94]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

In [95]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

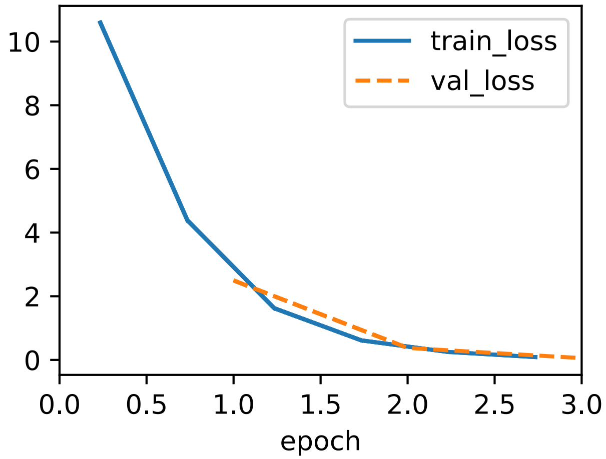
In [96]:
```
class SGD(d2l.HyperParameters):  #@save # Our Stochastic Gradient Descent
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

In [97]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

In [98]:
```
@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    return batch

@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()

    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
```

```
            self.optim.zero_grad()
            with torch.no_grad():
                loss.backward()
                if self.gradient_clip_val > 0:  # To be discussed later
                    self.clip_gradients(self.gradient_clip_val, self.model)
                self.optim.step()
            self.train_batch_idx += 1

        if self.val_dataloader is None:
            return

        self.model.eval()
        for batch in self.val_dataloader:
            with torch.no_grad():
                self.model.validation_step(self.prepare_batch(batch))
            self.val_batch_idx += 1
```

In [99]:
```
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 [100…
```
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([ 0.1287, -0.2016])
error in estimating b: tensor([0.2353])
```

4.2. The Image Classification Dataset

In [101…
```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

4.2.1. Loading the Dataset

In [102…
```
class FashionMNIST(d2l.DataModule):  #@save
    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 [103…
```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

Out[103…
```
(60000, 10000)
```

In [104…
```
data.train[0][0].shape
```

Out[104…
```
torch.Size([1, 32, 32])
```

In [105…
```
@d2l.add_to_class(FashionMNIST)  #@save
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]
```

4.2.2. Reading a Minibatch

In [106…

```
@d2l.add_to_class(FashionMNIST)  #@save
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 [107…

```
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 [108…

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

Out[108…

```
'2.15 sec'
```

4.2.3. Visualization

In [109…

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    raise NotImplementedError
```

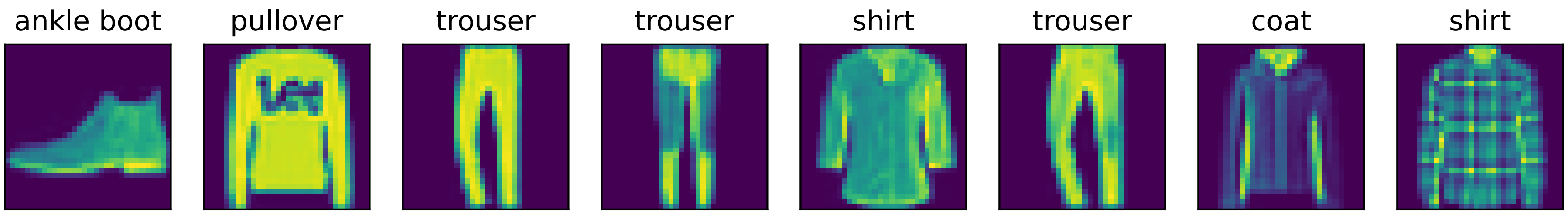
In [110…

```
@d2l.add_to_class(FashionMNIST)  #@save
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)
```

In [111…

```
batch = next(iter(data.val_dataloader()))
```

```
data.visualize(batch)
```



4.3. The Base Classification Model

In [112…

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

4.3.1. The Classifier Class

In [113…

```
class Classifier(d2l.Module):  #@save
    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 [114…

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

In [115…

```
@d2l.add_to_class(Classifier)  #@save
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 [116…

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

4.4.1. The Softmax

In [118…

```
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[118…

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

In [119…

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

In [120…

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

Out[120…

```
(tensor([[0.2064, 0.1756, 0.1952, 0.2390, 0.1838],
         [0.2992, 0.2831, 0.1376, 0.1401, 0.1399]]),
 tensor([1.0000, 1.0000]))
```

4.4.2. The Model

In [121…

```
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 [122…
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

4.4.3. The Cross-Entropy Loss

In [123…
```
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]
```

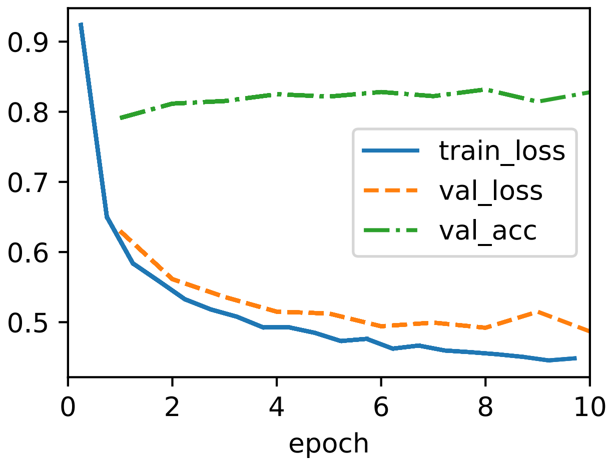
Out[123…
```
tensor([0.1000, 0.5000])
```

In [124…
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[124…
```
tensor(1.4979)
```

In [125…
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

In [126…
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)

trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

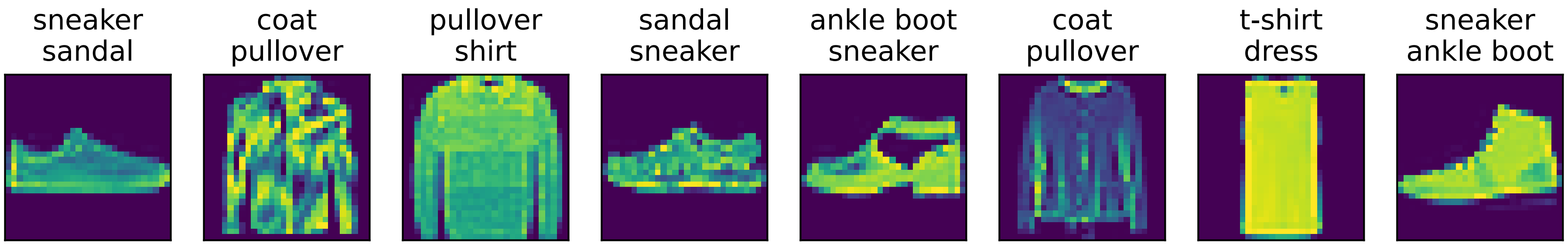


4.4.5. Prediction

In [127…
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[127…
```
torch.Size([256])
```

In [128…
```
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. Multilyaer Perceptrons

In [129…

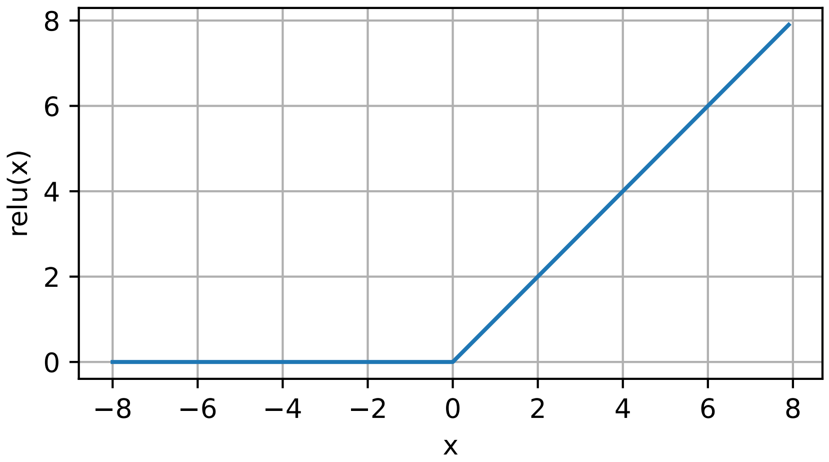
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

5.1.2. Activation Functions

5.1.2.1. ReLU Function

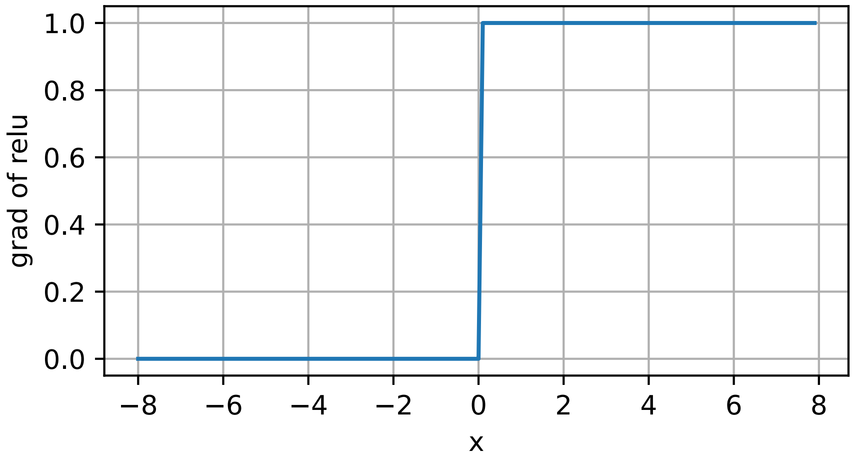
In [130…

```
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 [131…

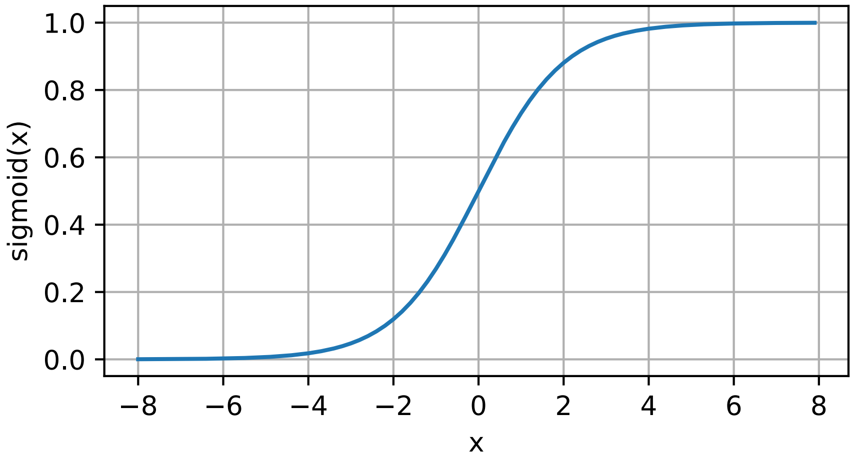
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



5.1.2.2. Sigmoid Function

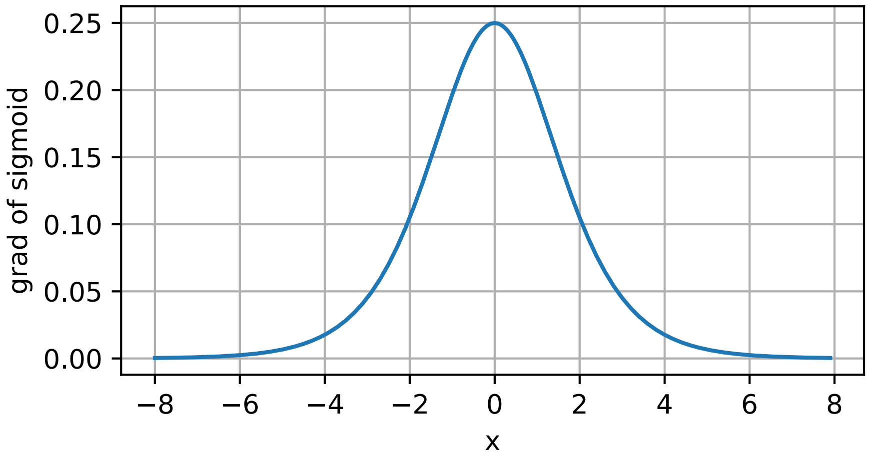
In [132…

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



In [133…

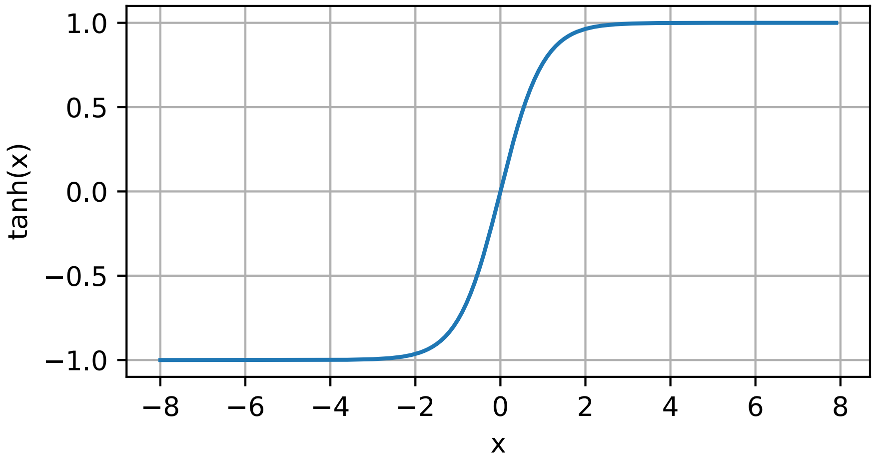
```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



5.1.2.3. Tanh Function

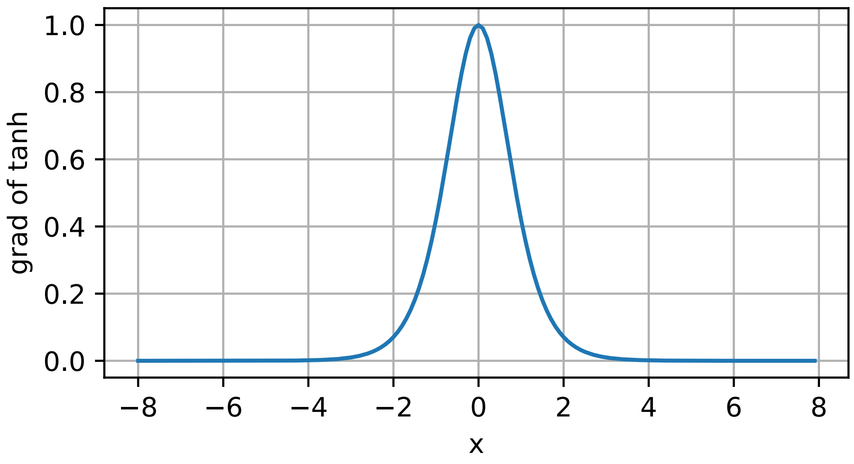
In [134…

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



In [135…

```
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 [136…

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

5.2.1. Implementation from Scratch

5.2.1.1. Initializing Model Parameters

In [137…

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

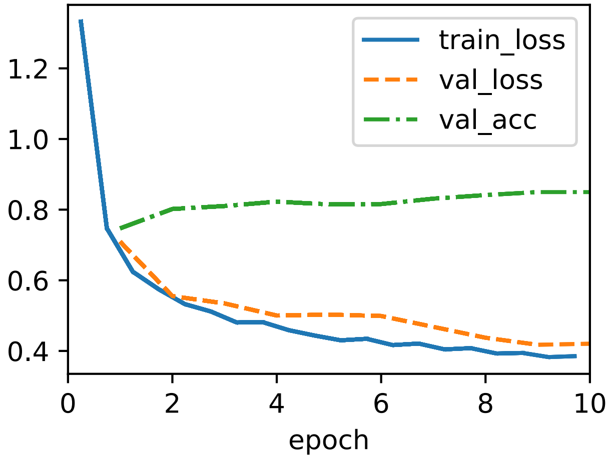
5.2.1.2. Model

In [138…
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

In [139…
```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

5.2.1.3. Training

In [140…
```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



5.2.2. Concise Implementation

5.2.2.1. Model

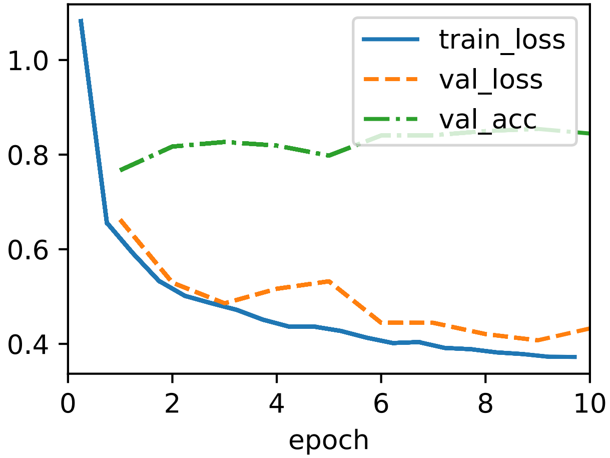
In [141…

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

5.2.2.2. Training

In [142…

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



Discussion

2.1. Data Manipulation

2.1.5. Saving Memory

To prevent memory waste, we should use slice notation [:] rather than just using variables!

For example,
A = A + B (X)
A[:] += A + B (O)

2.2. Data Preprocessing

2.2.2. Data Preparation

We can get rid of NaN values, which might be dangerous when we use data, by introducing some strategies(RoofType_Slate, RoofType_nan, or numerical values)!

2.3. Linear Algebra

2.3.5. Basic Properties of Tensor Arithmetics

We should not confuse * operator with matrix multiplication in tensor arithmetics! * operator is sort of scalar product...

2.3.6. Reduction

axis=0 is along columns, on the other hand, axis=1 is along rows!

2.3.8. Dot Product

* operator can be considered as dot product in PyTorch!

2.3.9. Matrix-Vector Products

@ operator can be considered as matrix multiplication in PyTorch!

2.3.11. Norms

Sort of norms can be distinguished to three main norms!

1. torch.norm() (Vector) : Euclidean norm
2. abs().sum() : Manhattan distance
3. torch.norm() (Matrix) : Frobenius norm

2.5. Automatic Differentiation

2.5.1. A Simple Function

- If y is scalar function of vector x, we can get gradient of y(x.grad) by processing y.backward().
- If we want to reset gradient of y, we can run x.grad.zero_().

2.5.2. Backward for Non-Scalar Variables

- If y is not a scalar but a vector, we can earn Jacobian derivatives.
- However, we generally get summing up the gradients of each component of y, w.r.t. vector x.

2.5.3. Detaching Computation

- If we want to use value of vector x to express other variables but not want to consider those variables as function of x, we can copy the value of x to other new variables, where we can avoid to be differentiated by x, by using detach().

3.1. Linear Regression

3.1.1. Basics

- Minibatch SGD method is widely used in Deep Learning, but why? Quasi-Newton method might do better performance?
- Maybe the local minimum problem cannot be dealt with in Quasi-Newton, while SGD can be...

3.1.2. Vectorization for Speed

- In deep learning, especially training phase, we should use vectorization method rather than for-loop to utilize running time!

3.2. Object-Oriented Design for Implementation

3.2.1. Utilities

- @add_to_class() : We can add specific function to class after the class is created, even after instances are generated!
- @class HyperParameters : We can add all arguments in **init** method to class attributes!

4.1. Softmax Regression

4.1.1. Classification

- Regression cannot deal with all problem!
- In classification problem, we focus on "which category?" questions. Indeed, there are cases where more than one label might be true!
- There might be problems such as the probability might exceed 1 when it comes to the linear model...
- To solve the problems, we should normalize all probabilities between 0 and 1, and the sum of probabilities is always 1. We call the function, which take on the role, Softmax function!
- To improve computational efficiency as well, we can vectorize data!

4.1.2. Loss Function

In Softmax regression, we can define loss function by using log-likelihood, where cross-entropy is introduced!

4.2. The Image Classification Dataset

4.2.2. Reading a Minibatch

We can use built-in data iterator rather than creating on our own, by using iter(data.train_dataloader())!

4.3. The Base Classification Model

4.3.2. Accuracy

When we define accuracy() function to determine what label is most accurate to given data, we should match data type between y_hat and y because == operator is sensitive to data type!

4.4. Softmax Regression Implementation from Scratch

4.4.1. The Softmax

If we implement softmax function by scratch, we must indicate that argument X is potentially dangerous when X is too small or too large!

4.4.3. The Cross-Entropy Loss

We can create loss function by introducing cross-entropy loss, which is general in deep learning! (However, this method we are currently implementing is just regression...)

5.1. Multilayer Perceptrons

To jump beyond the limitation of linear model, we can introduce hidden layers between linear layers, and nonlinear activation function such as ReLU, Sigmoid, tanh function!

5.2. Implementation of Multilayer Perceptrons

As a matter of fact, there are nothing to say as discussions... (This chapter are simply talking about how to implement MLP!)

5.3. Forward Propagation, Backward Propagation, and Computational Graphs

5.3.1. Forward Propagation

In computational graph, we can use forward propagation to compute and save intermediate variables, from input layer to output layer!

5.3.3. Backpropagation

To compute gradient of parameters, we should introduce backpropagation in neural network, using chain rule!

5.3.4. Training Neural Networks

There are dependencies on forward propagation and backward propagation. In other words, the forward propagation computes parameters traversing on the computational graph, and then, the backpropagation computes gradients of parameters to correct them, using chain rule!

Takeaway

Writing the codes, I could learn a lot of useful things.
Firstly, it was more hard than I thought to type the codes and think about what do those mean.
But I felt that as well it is also crucial to enhance my performance in this deep learning course, rather than just watching those materials.
Secondly, I could figure out that the difference of Python grammar between normal Python codes and PyTorch codes is slightly, but much bigger than I thought. That aspect made me feel slightly strange.