HW1

September 24, 2024

[75]:
```
import torch
```

1 2.1 Data Manipulation

[76]:
```
x = torch.arange(12, dtype=torch.float32)
x
```

[76]:
```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

[77]:
```
x.numel()
```

[77]:
```
12
```

[78]:
```
x.shape
```

[78]:
```
torch.Size([12])
```

[79]:
```
X = x.reshape(3,4)
X
```

[79]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

[80]:
```
X.shape
```

[80]:
```
torch.Size([3, 4])
```

[81]:
```
Y = x.reshape(3,-1)
Y
```

[81]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

[82]:
```
Y.shape
```

[82]:
```
torch.Size([3, 4])
```

[83]:
```
torch.zeros((2,3,4))
```

[83]:
```
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.]]])
```

[84]:
```
torch.ones((2,3,4))
```

[84]:
```
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.]]])
```

[85]:
```
torch.randn(3, 4)
```

[85]:
```
tensor([[ 0.7569, -0.8846,  1.2242,  1.1496],
        [ 2.4280, -0.9975, -0.7236, -0.2734],
        [ 2.3999,  0.1009, -0.6625,  0.0480]])
```

[86]:
```
torch.randn(3, 4)
```

[86]:
```
tensor([[ 0.0846,  0.0951,  0.5754,  0.1748],
        [-0.6192,  1.2792,  0.5654, -2.6387],
        [-0.2107, -1.1680, -1.1574,  0.2013]])
```

[87]:
```
X[-1], X[1:3]
```

[87]:
```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

[88]:
```
X[1, 2] = 17
X
```

[88]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

[89]:
```
X[:2, :] = 12
X
```

[89]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

[90]:
```
torch.exp(x)
```

[90]:
```
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])
```

[91]:
```
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
```

[91]:
```
(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.]))
```

[92]:
```
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
```

[92]:
```
(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.]))
```

[93]:
```
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)
```

[93]:
```
(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.]]))
```

[94]:
```
X == Y
```

[94]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
```

```
        [False, False, False, False]])
```

[95]:
```
X.sum()
```

[95]:
```
tensor(66.)
```

[96]:
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

[96]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

[97]:
```
a + b
```

[97]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

[98]:
```
before = id(Y)
Y = Y + X
id(Y) == before
```

[98]:
```
False
```

[99]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 2702656993936
id(Z): 2702656993936
```

[100]:
```
before = id(X)
X += Y
id(X) == before
```

[100]:
```
True
```

[101]:
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

[101]:
```
(numpy.ndarray, torch.Tensor)
```

[102]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

[102]: (tensor([3.5000]), 3.5, 3.5, 3)

[103]:
```
X<Y, X>Y
```

[103]:
```
(tensor([[False, False, False, False],
         [False, False, False, False],
         [False, False, False, False]]),
 tensor([[False,  True,  True,  True],
         [ True,  True,  True,  True],
         [ True,  True,  True,  True]]))
```

[104]:
```
c = torch.arange(60).reshape((3, 4, 5))
a+c
```

[104]:
```
tensor([[[ 3.5000,  4.5000,  5.5000,  6.5000,  7.5000],
         [ 8.5000,  9.5000, 10.5000, 11.5000, 12.5000],
         [13.5000, 14.5000, 15.5000, 16.5000, 17.5000],
         [18.5000, 19.5000, 20.5000, 21.5000, 22.5000]],

        [[23.5000, 24.5000, 25.5000, 26.5000, 27.5000],
         [28.5000, 29.5000, 30.5000, 31.5000, 32.5000],
         [33.5000, 34.5000, 35.5000, 36.5000, 37.5000],
         [38.5000, 39.5000, 40.5000, 41.5000, 42.5000]],

        [[43.5000, 44.5000, 45.5000, 46.5000, 47.5000],
         [48.5000, 49.5000, 50.5000, 51.5000, 52.5000],
         [53.5000, 54.5000, 55.5000, 56.5000, 57.5000],
         [58.5000, 59.5000, 60.5000, 61.5000, 62.5000]]])
```

2 2.1 Data Preprocessing

[105]:
```
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''')
```

[106]:
```
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
```

[107]:
```
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
```

[108]:
```
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
```

[109]:
```
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

[109]:
```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

3 2.3 Linear Algebra

[110]:
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

[110]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

[111]:
```
x = torch.arange(3)
x
```

[111]:
```
tensor([0, 1, 2])
```

[112]:
```
x[2]
```

[112]:
```
tensor(2)
```

[113]:
```
len(x)
```

[113]:
```
3
```

[114]:
```
x.shape
```

[114]:
```
torch.Size([3])
```

[115]:
```
A = torch.arange(6).reshape(3, 2)
A
```

[115]:
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

[116]:
```
A.T
```

[116]:
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

[117]:
```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

[117]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

[118]:
```
torch.arange(24).reshape(2, 3, 4)
```

[118]:
```
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]]])
```

[119]:
```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()  # Assign a copy of A to B by allocating new memory
A, A + B
```

[119]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
```

```
        [ 6.,  8., 10.]]))
```

[120]:
```
A * B
```

[120]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

[121]:
```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

[121]:
```
(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]))
```

[122]:
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

[122]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

[123]:
```
A.shape, A.sum()
```

[123]:
```
(torch.Size([2, 3]), tensor(15.))
```

[124]:
```
A.shape, A.sum(axis=0).shape
```

[124]:
```
(torch.Size([2, 3]), torch.Size([3]))
```

[125]:
```
A.shape, A.sum(axis=1).shape
```

[125]:
```
(torch.Size([2, 3]), torch.Size([2]))
```

[126]:
```
A.sum(axis=[0, 1]) == A.sum()  # Same as A.sum()
```

[126]:
```
tensor(True)
```

[127]:
```
A.mean(), A.sum() / A.numel()
```

[127]:
```
(tensor(2.5000), tensor(2.5000))
```

[128]:
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

[128]:
```
(tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

[129]:
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

[129]:
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

[130]:
```
A / sum_A
```

[130]:
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

[131]:
```
A.cumsum(axis=0)
```

[131]:
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

[132]:
```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

[132]:
```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

[133]:
```
torch.sum(x * y)
```

[133]:
```
tensor(3.)
```

[134]:
```
A.shape, x.shape, torch.mv(A, x), A@x
```

[134]:
```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))
```

[135]:
```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

[135]:
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

[136]:
```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

[136]:
```
tensor(5.)
```

[137]:
```
torch.abs(u).sum()
```

[137]:
```
tensor(7.)
```

[138]:
```
torch.norm(torch.ones((4, 9)))
```

[138]: tensor(6.)

4 2.5 Automatic Differentiation

[139]:
```python
x = torch.arange(4.0)
x
```

[139]: tensor([0., 1., 2., 3.])

[140]:
```python
# Can also create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad  # The gradient is None by default
```

[141]:
```python
y = 2 * torch.dot(x, x)
y
```

[141]: tensor(28., grad_fn=<MulBackward0>)

[142]:
```python
y.backward()
x.grad
```

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

[143]:
```python
x.grad == 4 * x
```

[143]: tensor([True, True, True, True])

[144]:
```python
x.grad.zero_()  # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

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

[145]:
```python
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```

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

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

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

[146]:
```python
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x
```

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

[146]:
```
tensor([True, True, True, True])
```

[147]:
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

[147]:
```
tensor([True, True, True, True])
```

[148]:
```
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
```

[149]:
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

[150]:
```
a.grad == d / a
```

[150]:
```
tensor(True)
```

5 3.1 Linear Regression

[151]:
```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

[152]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

[153]:
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
```

```
f'{time.time() - t:.5f} sec'
```

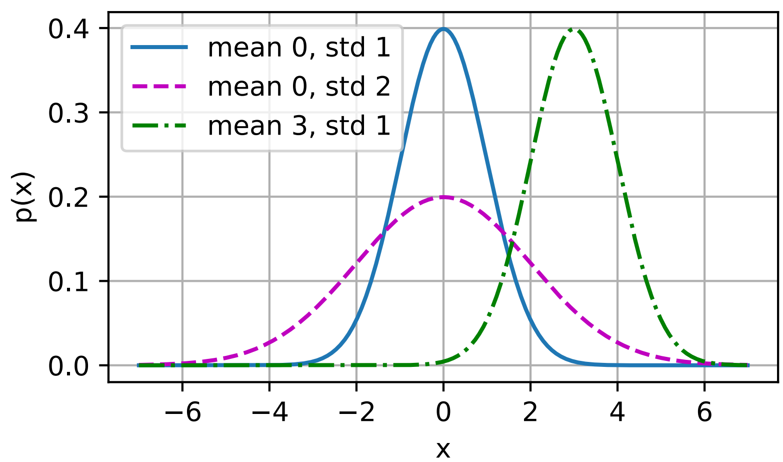
[153]: '0.11298 sec'

[154]:
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

[154]: '0.00000 sec'

[155]:
```
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)
```

[156]:
```
# Use NumPy again for visualization
x = np.arange(-7, 7, 0.01)

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



6 3.2 Object-Oriented Design for Implementation

[157]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

[158]:
```
def add_to_class(Class):  #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

[159]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

[160]:
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

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

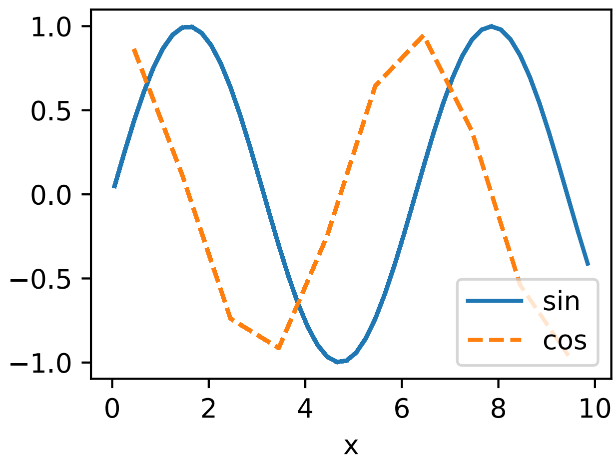
[161]:
```
class HyperParameters:  #@save
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

[162]:
```
# Call the fully implemented HyperParameters class saved in d2l
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))

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

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

[163]:
```
class ProgressBoard(d2l.HyperParameters):  #@save
    """The board that plots data points in animation."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2', 'C3'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

[164]:
```
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)
```



[165]:
```
class Module(nn.Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
        super().__init__()
        self.save_hyperparameters()
        self.board = ProgressBoard()

    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
```

```
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

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

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

    def configure_optimizers(self):
        raise NotImplementedError
```

[166]:
```
class DataModule(d2l.HyperParameters):  #@save
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

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

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

[167]:
```
class Trainer(d2l.HyperParameters):  #@save
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

    def prepare_model(self, model):
        model.trainer = self
```

```python
        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
```

7 3.4 Linear Regression Implementation from Scratch

[168]:
```python
%matplotlib inline
import torch
from d2l import torch as d2l
```

[169]:
```python
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

[170]:
```python
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

[171]:
```python
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

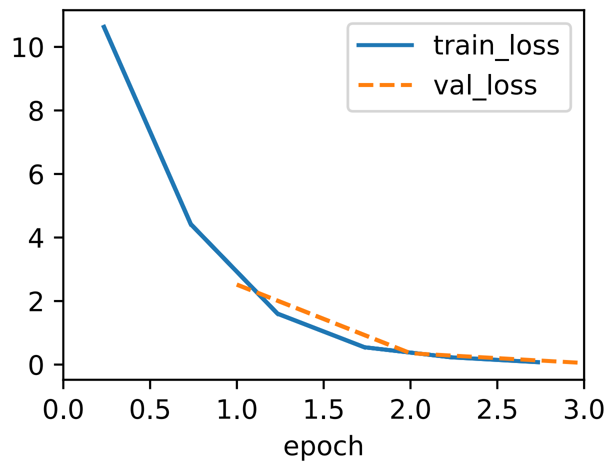
[172]:
```python
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
```

```
            param -= self.lr * param.grad

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

[173]:
```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

[174]:
```
@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
```

[175]:
```
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)
```



[176]:
```python
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([ 0.1100, -0.1897])
error in estimating b: tensor([0.2141])
```

8 4.1 Softmax Regression

9 4.2 The Image Classification Dataset

[177]:
```python
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

[178]:
```python
class FashionMNIST(d2l.DataModule):  #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
```

```
                root=self.root, train=False, transform=trans, download=True)
```

[179]:
```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

[179]: (60000, 10000)

[180]:
```
data.train[0][0].shape
```

[180]: torch.Size([1, 32, 32])

[181]:
```
@d2l.add_to_class(FashionMNIST)  #@save
def text_labels(self, indices):
    """Return text labels."""
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

[182]:
```
@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)
```

[183]:
```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

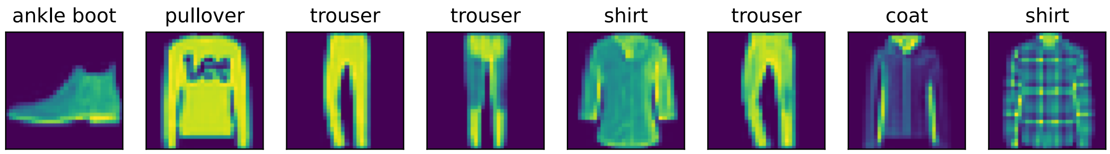
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64

[184]:
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

[184]: '6.09 sec'

[185]:
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    raise NotImplementedError
```

[186]:
```
@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)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```



10 4.3 The Base Classification Model

[187]:
```
import torch
from d2l import torch as d2l
```

[188]:
```
class Classifier(d2l.Module):  #@save
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)
```

[189]:
```
@d2l.add_to_class(d2l.Module)  #@save
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

[190]:
```
@d2l.add_to_class(Classifier)  #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the number of correct predictions."""
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

11 4.4 Softmax Regression Implementation from Scratch

[191]:
```
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)
```

[191]:
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

[192]:
```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

[193]:
```python
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

[193]:
```
(tensor([[0.2147, 0.2238, 0.0983, 0.2420, 0.2212],
         [0.1714, 0.2793, 0.1708, 0.2187, 0.1598]]),
 tensor([1., 1.]))
```

[194]:
```python
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]
```

[195]:
```python
@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)
```

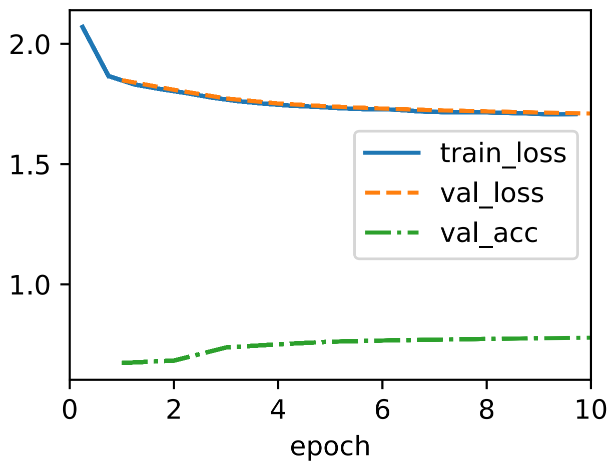
[196]:
```python
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

[196]:
```
tensor([0.1000, 0.5000])
```

[197]:
```python
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

[197]:
```
tensor(1.4979)
```

[198]:
```python
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)
```



[199]:
```python
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

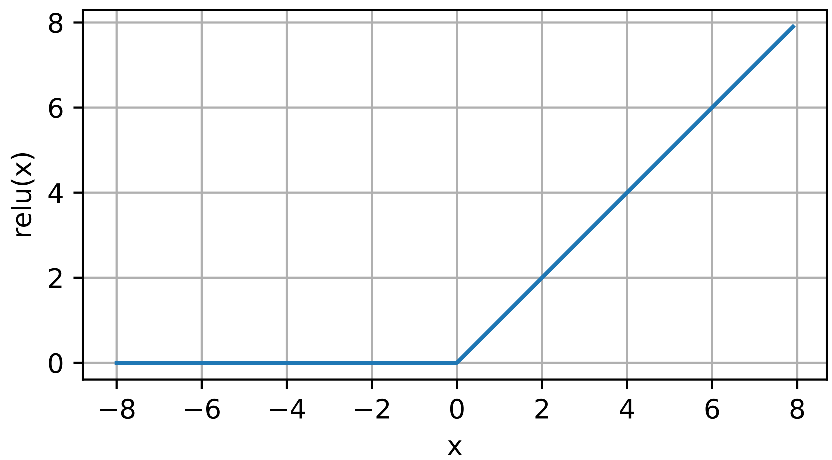
[199]:
```
torch.Size([256])
```

[200]:
```python
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)
```

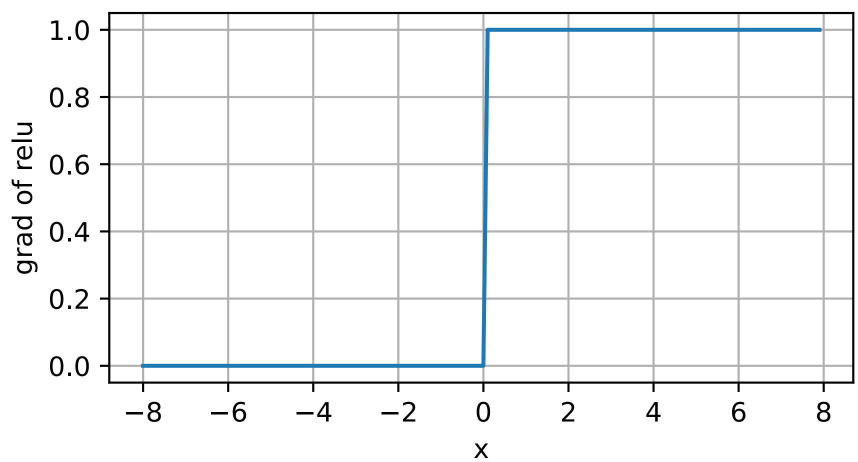
12 5.1 Multilayer Perceptrons

[201]:
```python
%matplotlib inline
import torch
from d2l import torch as d2l
```

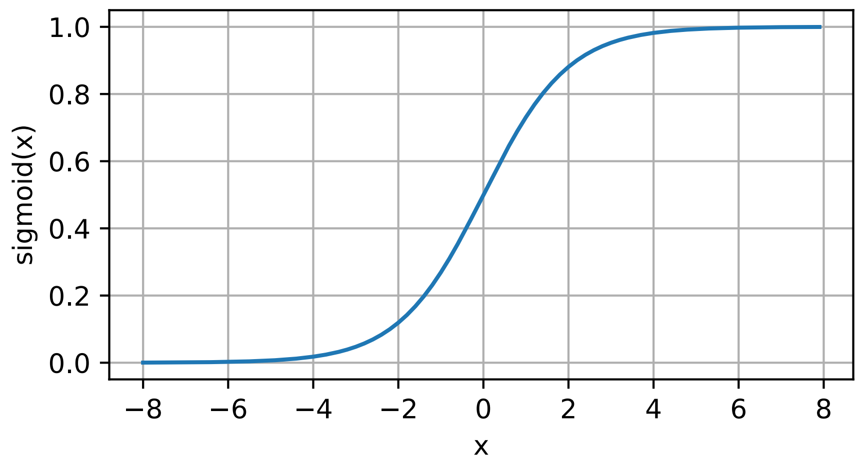
[202]:
```python
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



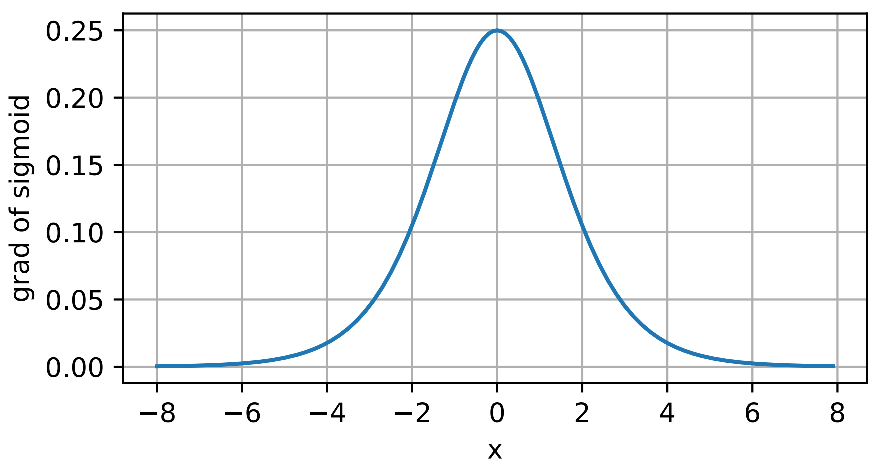
[203]:
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



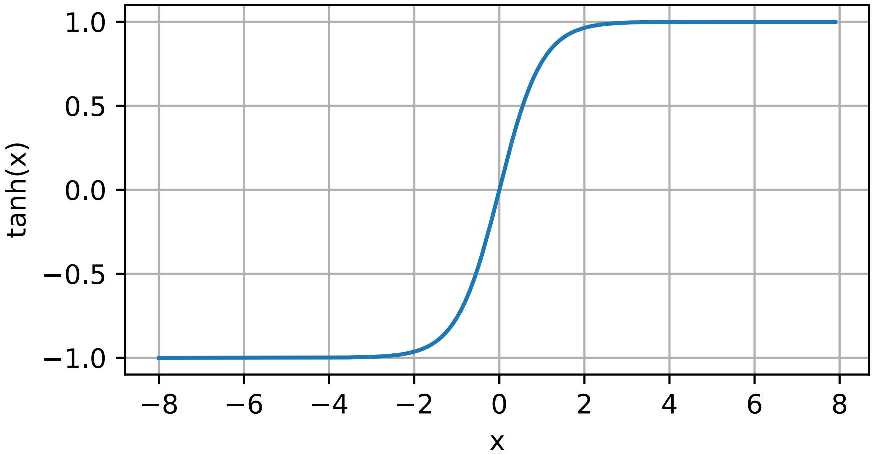
[204]:
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



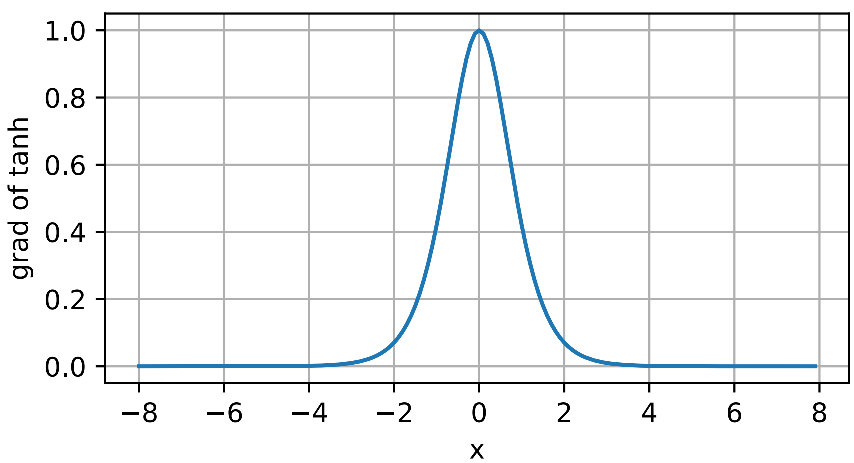
[205]:
```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



[206]:
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



[207]:
```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



13 5.2 Implementation of Multilayer Perceptrons

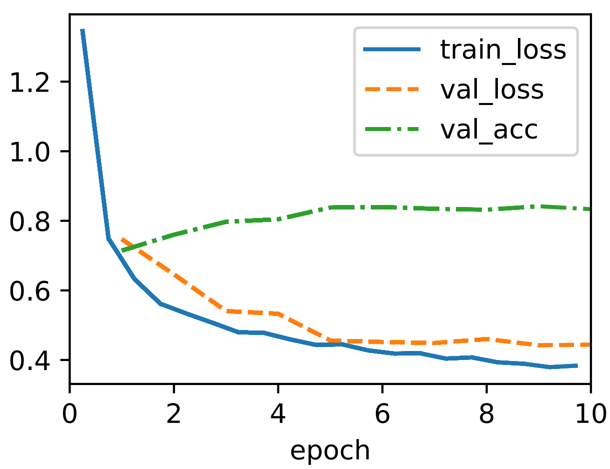
[208]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

[209]:
```
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))
```

[210]:
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

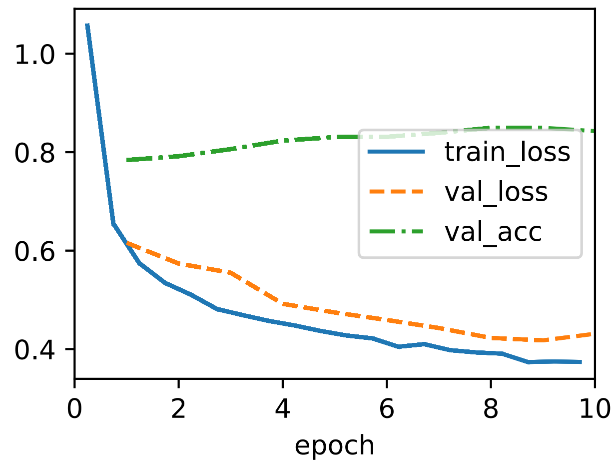
[211]:
```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

[212]:
```
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)
```



[213]:
```
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))
```

[214]:
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



14 5.3 Forward Propagation, Backward Propagation and Computational Graphs

Discussion:

In neural networks, **forward propagation** is the process where input data $\mathbf{x}$ passes through the network layers to compute the output. Each layer transforms the input using weights $\mathbf{W}$ and activation functions. For a neural network with one hidden layer, the hidden activations are computed as:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \quad \mathbf{h} = \phi(\mathbf{z})$$

where $\mathbf{W}^{(1)}$ are the hidden layer weights and ϕ is the activation function. The output layer is computed similarly:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}$$

Backpropagation is used to calculate the gradient of the loss function with respect to the network parameters. This is done by applying the chain rule of calculus. For example, the gradient of the objective function J with respect to the weights $\mathbf{W}^{(2)}$ in the output layer is computed as:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

This process continues backward through the layers to update all the weights of the network.

15 Discussion and Self Excercise

15.1 3.4 Linear Regression from Scratch

Use different Loss Function

[233]:
```python
def loss(self, y_hat, y):
    l = (y_hat - d2l.reshape(y, y_hat.shape)).abs().sum()  # Absolute value of␣
 ↪(y_hat - y)
    return l.mean()  # Return the mean absolute difference
```

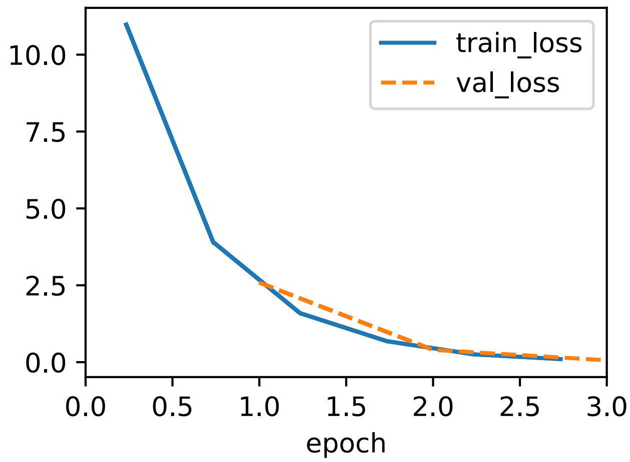
[234]:
```python
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

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

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

[235]:
```python
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

[236]:
```python
@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
```

[237]:
```
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)
```



[239]:
```
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([ 0.1082, -0.2167])
error in estimating b: tensor([0.2502])
```

Discussion: It has lower error in estimating weight but a higher error in estimating bias

15.2 4.1 Softmax Regression

Discussion:

In softmax regression, the model assigns probabilities to each possible class, ensuring that all probabilities sum to 1. The model uses the **softmax function** to convert raw output scores (logits) into probabilities. Given input ($\mathbf{x}$) and weights ($\mathbf{W}$), the predicted probability for class (j) is calculated as:

$$\hat{y}_j = \frac{\exp(\mathbf{w}_j^\top \mathbf{x})}{\sum_k \exp(\mathbf{w}_k^\top \mathbf{x})}$$

where ($\hat{y}_j$) is the predicted probability for class (j), ($\mathbf{w}_j$) is the weight vector for class (j), and the sum is over all possible classes (k).

15.3 Loss Function and Cross-Entropy

To measure how well the model's predicted probabilities ($\hat{\mathbf{y}}$) match the true class labels ($\mathbf{y}$), we use the **cross-entropy loss function**. For a single data point with true class label (y) and predicted probability distribution ($\hat{\mathbf{y}}$), the cross-entropy loss is:

$$L(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_j y_j \log(\hat{y}_j)$$

Here, (y_j) is a one-hot encoded vector indicating the true class (with 1 for the correct class and 0 for the others), and ($\hat{y}_j$) is the predicted probability for class (j). This loss function penalizes the model heavily when it assigns a low probability to the true class.

15.4 Entropy and Surprisal

The **entropy** ($H(\mathbf{p})$) of a probability distribution ($\mathbf{p}$) measures the uncertainty or "surprise" in a prediction. It is given by:

$$H(\mathbf{p}) = -\sum_j p_j \log(p_j)$$

The concept of **surprisal** quantifies how unexpected an event is. For a given event (j) with probability (p_j), the surprisal is:

$$S(p_j) = -\log(p_j)$$

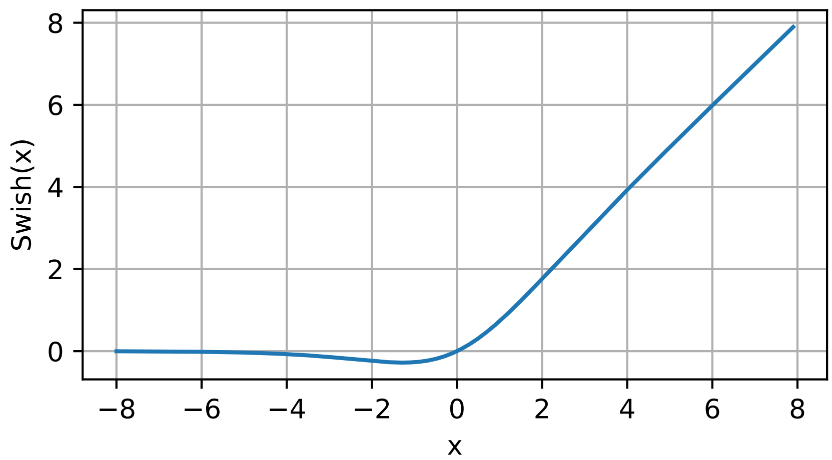
In softmax regression, minimizing the cross-entropy loss essentially reduces the "surprise" of the model's predictions by making them more accurate and aligned with the actual labels.

By optimizing the model to minimize the cross-entropy loss, we effectively reduce the overall surprisal, ensuring the predicted probabilities are as close as possible to the true distribution.

15.5 5.1 MLP

Swish Activation Function:

[216]:
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = x * torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'Swish(x)', figsize=(5, 2.5))
```



pReLU Activation Function:

[218]:
```
alpha = 0.2
y = torch.max(torch.zeros_like(x), x) + alpha * torch.min(torch.zeros_like(x),␣
 ↪x)
d2l.plot(x.detach(), y.detach(), 'x', 'pReLU(x)', figsize=(5, 2.5))
```