2.1-Data_Manipulation

September 26, 2024

0.1 2.1.Data Manipulation

[2]:
```
import torch
```

[3]:
```
x=torch.arange(12,dtype=torch.float32)
x
```

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

[4]:
```
x.numel()
```

[4]:
```
12
```

[5]:
```
x.shape
```

[5]:
```
torch.Size([12])
```

[6]:
```
X=x.reshape(3,4)
X
```

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

[7]:
```
Y=x.reshape(-1,6)
Y
```

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

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

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

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

[9]:
```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

[10]:
```
torch.ones((3,4),dtype=torch.int64)
```

[10]:
```
tensor([[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]])
```

[11]:
```
torch.tensor([[2,1,4,3],[1,2,3,4],[4,3,2,1]])
```

[11]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

[12]:
```
X[-1]
```

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

[13]:
```
X[1:3]
```

[13]:
```
tensor([[ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

[14]:
```
X[1,2] =17
X
```

[14]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

[15]:
```
X[:2,:]=12
X
```

[15]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

[16]:
```
torch.exp(x)
```

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

[17]:
```
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
```

[17]:
```
(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.]))
```

[18]:
```
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)
# dim indicates axis. if dim=0, sum along axis 0.
```

[18]:
```
(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.]]))
```

[19]:
```
X==Y
```

[19]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

[20]:
```
X.sum()
```

[20]:
```
tensor(66.)
```

[21]:
```
a=torch.arange(3).reshape((3,1))
b=torch.arange(2).reshape((1,2))
a,b
```

[21]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

[22]:
```
a+b
#a duplicates its column, b duplicates its row to make themselves in torch.
 ↪Size([3,2])
```

[22]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

[23]:
```
before = id(Y)
Y = X+Y
id(Y) == before
# Y assigned X+Y, but in fact new variable was created and named Y.
# Memory used a lot.
```

[23]:
```
False
```

[24]:
```
Z=torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X+Y # a way to assign value to existing variable (in-place) - use␣
↪'slicing'
print('id(Z):', id(Z))
```

```
id(Z): 1850606144592
id(Z): 1850606144592
```

[25]:
```
before = id(X)
X += Y # in-place
id(X) == before
```

[25]:
```
True
```

[26]:
```
A = X.numpy()
B = torch.from_numpy(A) # make ndarray into torch.tensor, by this ndarray and␣
↪torch.tensor shares the same memory.
type(A),type(B)
```

[26]:
```
(numpy.ndarray, torch.Tensor)
```

[27]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a) # convert tensor with one value into scalar - 1. .
↪item, 2.
```

[27]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

2.2-Data_Processing

September 26, 2024

[20]:
```
import os

os.makedirs(os.path.join('data'), exist_ok=True) # create folder in path "../
↪data"
data_file = os.path.join('data', 'house_tiny.csv') # create file in path "../
↪data/house_tiny.csv"
with open(data_file, 'w') as f: # open file in writing mode
    f.write('''NumRooms,RoofType,Price
NA,NA,127500
2,NA,106000
4,Slate,178100
NA,NA,140000''')
```

[21]:
```
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
```

1. How to preprocess NaN value

[22]:
```
inputs, targets = data.iloc[:,0:2], data.iloc[:,2] # indexing columns of tabel␣
↪data by index using "iloc"
inputs = pd.get_dummies(inputs, dummy_na=True) # split Column which has NaN␣
↪values into isNaN column and is<other-features> columns, each columns only␣
↪have true and false value.
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
```

[23]:
```
inputs = inputs.fillna(inputs.mean()) # replace NaN with other values' mean
↪(imputation)
print(inputs)
```

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

2. Transform dataframe into tensor dataframe -> ndarray -> tensor

[24]:
```
import torch

X=torch.tensor(inputs.to_numpy(dtype=float)) # transfrom dataFrame ndarray, and
↪turn it into tensor.
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

[24]:
```
(tensor([[3., 0., 1.],
         [2., 0., 1.],
         [4., 1., 0.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

[43]:
```
url = "https://archive.ics.uci.edu/static/public/1/data.csv"
data = pd.read_csv(url)

data.head(10)

sex = data.loc[data['Sex']=='M','Sex']
sex
```

[43]:
```
0       M
1       M
3       M
8       M
11      M
       ..
4170    M
4171    M
4173    M
4174    M
4176    M
Name: Sex, Length: 1528, dtype: object
```

2.3-Linear_Algebra

September 26, 2024

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

[2]:
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

[2]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

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

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

[4]:
```
x[2]
```

[4]:
```
tensor(2)
```

[5]:
```
len(x)
```

[5]:
```
3
```

[6]:
```
x.shape
```

[6]:
```
torch.Size([3])
```

[7]:
```
A=torch.arange(6).reshape(3,2)
```

[8]:
```
A.T
```

[8]:
```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

[10]:
```
A = torch.tensor([[1,2,3],[2,0,4],[3,4,5]])
A == A.T
```

[10]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

[11]:
```
torch.arange(24).reshape(2,3,4)
```

[11]:
```
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]]])
```

[12]:
```
A = torch.arange(6, dtype=torch.float32).reshape(2,3)
B = A.clone()
A, A+B
```

[12]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

[13]:
```
A*B # In matrix '*' returns element-wise multiply
```

[13]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

[15]:
```
x=torch.arange(3, dtype=torch.float32)
x, x.sum()
```

[15]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

[18]:
```
A.shape, A.sum() #.sum() : return sum of all elements
```

[18]:
```
(torch.Size([2, 3]), tensor(15.))
```

[19]:
```
A.shape, A.sum(axis=0).shape # sum along certain axis, in result the tensor got␣
↪different dimension (reduction)
```

[19]:
```
(torch.Size([2, 3]), torch.Size([3]))
```

[20]:
```
A.shape, A.sum(axis=1).shape
```

[20]:
```
(torch.Size([2, 3]), torch.Size([2]))
```

[21]:
```
A.sum(axis=[0,1]) == A.sum()
```

[21]:
```
tensor(True)
```

[24]:
```
A.mean(), A.sum() / A.numel()
```

[24]: (tensor(2.5000), tensor(2.5000))

[25]:
```
A.mean(axis=0), A.sum(axis=0)/A.shape[0]
```

[25]: (tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))

[26]:
```
sum_A = A.sum(axis=1, keepdims = True)
sum_A, sum_A.shape
```

[26]:
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

[27]:
```
A / sum_A
```

[27]:
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

[28]:
```
A.cumsum(axis=0) # cumulative sum
```

[28]:
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

[30]:
```
y=torch.ones(3, dtype=torch.float32)
x,y,torch.dot(x,y) # dot : returns a sum of element-wise mul
```

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

[31]:
```
torch.sum(x*y) # same as dot
```

[31]: tensor(3.)

[32]:
```
A.shape, x.shape, torch.mv(A,x), A@x # @: can calculate matrix-matrix and␣
 ↪matrix-vector multiplication
```

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

[33]:
```
B=torch.ones(3,4)
torch.mm(A,B), A@B
```

[33]:
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

[36]:
```
u = torch.tensor([3.0, -4.0])
torch.norm(u) # norm2
```

[36]: tensor(5.)

[37]:
```
torch.abs(u).sum() # norm1
```

[37]: tensor(7.)

[39]:
```
torch.norm(torch.ones((4,9)))
```

[39]: tensor(6.)

2-5.Automation_Differentiation

September 26, 2024

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

[2]:
```
x=torch.arange(4.0)
x
```

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

[3]:
```
x.requires_grad_(True)
x.grad # make a vacant space for storing gradient of x
```

[4]:
```
y=2*torch.dot(x,x)
y
```

[4]:
```
tensor(28., grad_fn=<MulBackward0>)
```

[5]:
```
y.backward() # backpropagation about y
x.grad # gradient of x is calculated by backpropagation of y
```

[5]:
```
tensor([ 0.,  4.,  8., 12.])
```

[6]:
```
x.grad == 4*x
```

[6]:
```
tensor([True, True, True, True])
```

[7]:
```
x.grad.zero_() # to store a new gradient of x, should make it zero. If not, the␣
↪value is added to existing value in x.grad
y = x.sum()
y.backward()
x.grad
```

[7]:
```
tensor([1., 1., 1., 1.])
```

[8]:
```
x.grad.zero_()
y = x*x
y.backward(gradient=torch.ones(len(y))) # gradient == v. v turns gradient value␣
↪into scalar.
x.grad
```

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

[9]:
```
x.grad.zero_()
y=x*x
u=y.detach() # used when calculating gradient of certain value at specific step
 ↪or layer.
z=u*x # without .detach, backpropagating z, gradient of x should be calculated
 ↪not only by u itself, but also by y which is multiplication of x and x
z.sum().backward()
x.grad == u # with .detach, backpropagating z, gradient of x equals to u. Even
 ↪though u is calculated by multiplication of x and x
```

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

[10]:
```
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
```

[13]:
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

[14]:
```
a.grad == d/a
```

[14]: tensor(True)

3-1_Linear_Regression

September 25, 2024

Regression : Used when predicting certain value. Especially linear regression is used when the value can be predicted via affine function.

When calculating the equation used is below. yhat = Xw + b X represents the whole datasets.

in yhat = wx + b, variable x represents only one dataset.

Loss function = 1/2*(yhat-y)**2 Square has benefit of lowering the difference, but it comes worse when there are some abnormal data. 1/2, the coeffiecient is just for canceling out the exponent when differentiating.

Mini batch : randomly sample a minibatch in certain size, average the loss on the mini-batch learnig-rate / batch-size : can be optimized by Bayesian optimization

minimizing MSE is equivalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

Question: what is a meaning of "minimizing MSE is equivalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise."

[2]:
```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

[3]:
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

[4]:
```
c = torch.zeros(n)
t = time.time()
for i in range(n): #for iteration is much slower
    c[i] = a[i] + b[i]
f'{time.time()-t:.5f} sec'
```

[4]: '0.53942 sec'

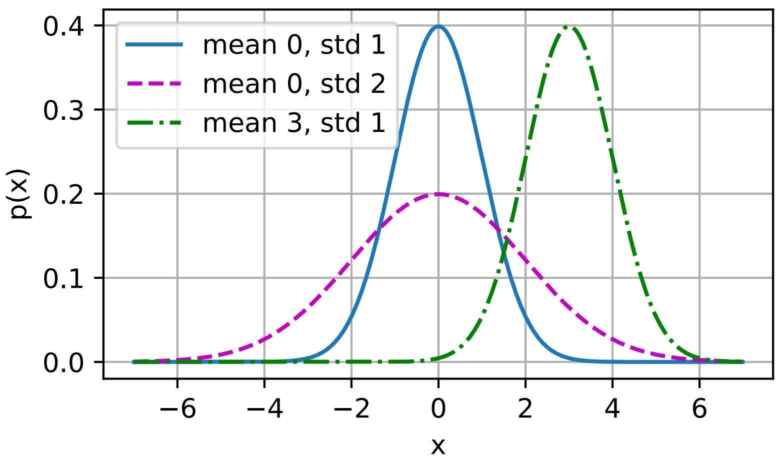
[5]:
```
t = time.time()
d=a+b #vector can be summed without iteration
```

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

[5]: '0.00110sec'

[6]:
```
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)
```

[7]:
```
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

September 25, 2024

[2]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

[3]:
```
# adding a method to a class, even if the instance of the class influenced by␣
↪this and gains the method.
def add_to_class(Class):
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

[4]:
```
class A:
    def __init__(self):
        self.b = 1
a=A()
```

[5]:
```
#use a decorator to add "do" method to class A
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do() #even if the instance of A created, it gains the method.
```

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

[6]:
```
# making hyperparameter arguments of a constructor into properties of the class
class HyperParameters:
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

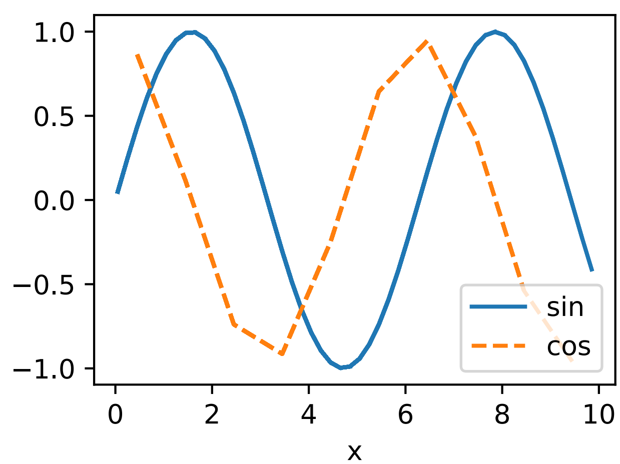
[7]:
```
# Inheriting d2l.HyperParameters, when calling save_hyperparameters, argument a␣
↪and b are saved as properties without statements. c is not saved due to␣
↪'ignore'
class B(d2l.HyperParameters):
    def __init__(self, a,b,c):
        self.save_hyperparameters(ignore=['c'])
```

```
        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
```

[8]:
```
# plot (x,y) and make a progress graph
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
```

[9]:
```
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)
```



[10]:
```
# Super class of all modules.
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 defined'
        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
```

[11]:
```
# Calling and preprocessing data
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)
```

```python
    # called to get validation data
    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

[12]:

```python
# Calling data for training
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.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

September 25, 2024

[11]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

1. parameters initialize

[12]:
```
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) #weight initialize by random numbers
        self.b = torch.zeros(1, requires_grad=True) #bias initialize as 0
```

2. defining model

[13]:
```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

3. loss function -> returning averaged loss value among all examples in minibatch

[14]:
```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

4. Optimization algorithm

[15]:
```
class SGD(d2l.HyperParameters):
    def __init__(self, params, lr):
        self.save_hyperparameters()

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

    # grad set to 0, must be called before backpropagation
    def zero_grad(self):
```

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

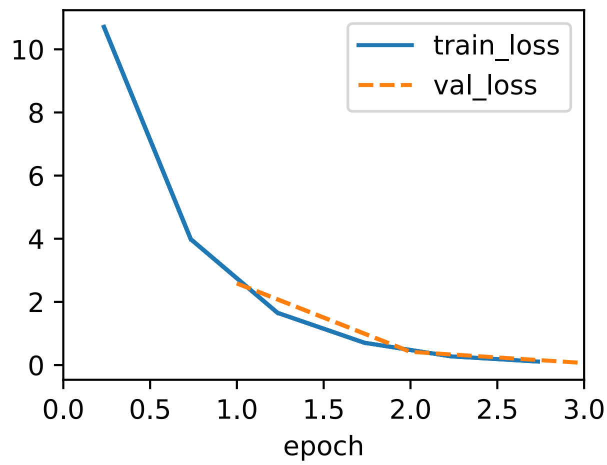
5. configure optimizer -> returning instance of SGD class

[16]:
```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

6. training

[17]:
```
@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()
    # train for each batch
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad(): # initializing grad
            loss.backward()
            if self.gradient_clip_val > 0:
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step() # param update with lr*grad
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    # foreach epoch, validate the model
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

[18]:
```
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)
#loss decreases
```



[19]:
```
with torch.no_grad():
    print(f'error in estimation w: {data.w - model.w.reshape(data.w.shape)}')
    print(f'error in estimating b: {data.b - model.b}')
```

```
error in estimation w: tensor([ 0.1261, -0.2299])
error in estimating b: tensor([0.2541])
```

4-1_Softmax_Regression

September 26, 2024

Classification:

hard : only classified as one category soft : can be classified as multi-category

Softmax function:

output is probability, and that probabiliy is a probability of for input value to be classified as certain category. Due to It is probability, the sum of sofftmax function is equal to 1, which is implemented by regularization. The output vecotr length must be equal to the number of the classes(categories)

Loss function:

1. Cross Entrophy
2. log likelihood

Q. why do we use Exponential power instead of using it's own value? To ignore small values?

4-2_The-Image-Classification-Dataset

September 25, 2024

[1]:
```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

[2]:
```
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)
```

[3]:
```
data = FashionMNIST(resize=(32, 32)) # convert image resolution into 32*32
len(data.train), len(data.val)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to ../data\FashionMNIST\raw\train-images-idx3-ubyte.gz

100.0%

Extracting ../data\FashionMNIST\raw\train-images-idx3-ubyte.gz to
../data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to ../data\FashionMNIST\raw\train-labels-idx1-ubyte.gz

100.0%
```

```
Extracting ../data\FashionMNIST\raw\train-labels-idx1-ubyte.gz to
../data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
../data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz

100.0%

Extracting ../data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz to
../data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
../data\FashionMNIST\raw\t10k-labels-idx1-ubyte.gz

100.0%

Extracting ../data\FashionMNIST\raw\t10k-labels-idx1-ubyte.gz to
../data\FashionMNIST\raw
```

[3]:
```
(60000, 10000)
```

[4]:
```
data.train[0][0].shape
```

[4]:
```
torch.Size([1, 32, 32])
```

[5]:
```
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    # label for classification categories
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal',␣
 ↪'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

[6]:
```
@d2l.add_to_class(FashionMNIST)
# parameter "train" for determining whether data is for train or validation
def get_dataloader(self, train):
    data = self.train if train else self.val
    # use built-in data iterator
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

[7]:
```
#using 'next' and 'iter' to get data
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

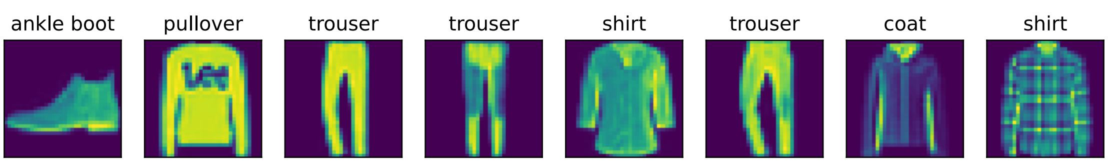
```
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

[8]:
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

[8]: '6.62 sec'

[9]:
```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    raise NotImplementedError
```

[10]:
```
@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.Base Classification Model

September 26, 2024

[2]:
```
import torch
from d2l import torch as d2l
```

[3]:
```
# Super class of all classifer modules
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)
```

[4]:
```
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

[5]:
```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    # returning the index of the argument of the highest probability, since we␣
 ↪use one-hot-incoding.
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

Question: 1. Since one-hot-encoding is too sparse, it is waste of space, why do not use binary number to represent words?

2. Since the wrong answer implies many information about the lack of performance of the model, why don't we grade wrong answers? For example, if the wrong answer is close to answer give 0.7 instead of uniformly give 0 to all kind of wrong answer. I believe we must make some criteria for evaluating similarity between a wrong answer and the correct one.

4-4_Softmax-Regression-Implementation-from-Scratch

September 26, 2024

[31]:
```
import torch
from d2l import torch as d2l
```

[32]:
```
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) # sum over specific axis
```

[32]:
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

[33]:
```
def softmax(X):
    X_exp = torch.exp(X) # as Softmax is a function of exp(x)/sum(exp(X))
    partition = X_exp.sum(1, keepdims=True) # denominator = partition function
    return X_exp / partition
```

Q. due to computational limit of python, isn't there no possibility of sum to not be 1? If so, does it matter?

[34]:
```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1) # when sum up, it must be 1. Softmax returns probability,␣
 ↪so it is natural for sum to be 1.
```

[34]:
```
(tensor([[0.2125, 0.1825, 0.2860, 0.1961, 0.1229],
         [0.1438, 0.2378, 0.2326, 0.2428, 0.1429]]),
 tensor([1.0000, 1.0000]))
```

[35]:
```
# output vector of softmax should have same length as the number of classes
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]
```

[36]:
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0])) #flatten
    return softmax(torch.matmul(X, self.W) + self.b)
```

[37]:
```
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] # y  indices
```

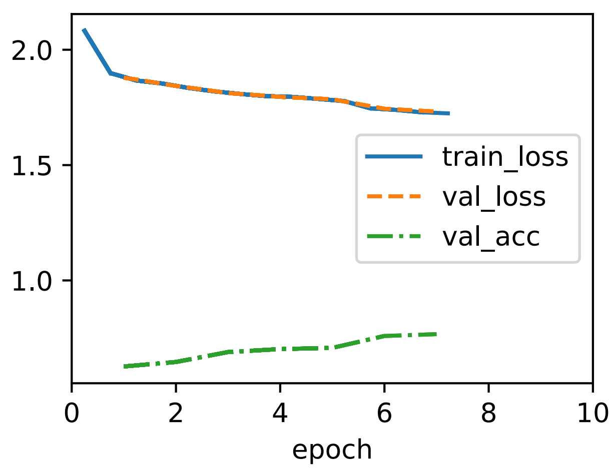
[37]:
```
tensor([0.1000, 0.5000])
```

[38]:
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean() # negative log likelihood

cross_entropy(y_hat, y)
```

[38]:
```
tensor(1.4979)
```

Q. how to optimize hyper parameters? How to select hyper params?

[ ]:
```
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)
```



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

[ ]:
```
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

September 26, 2024

1 5.1.Multilayer Perceptrons

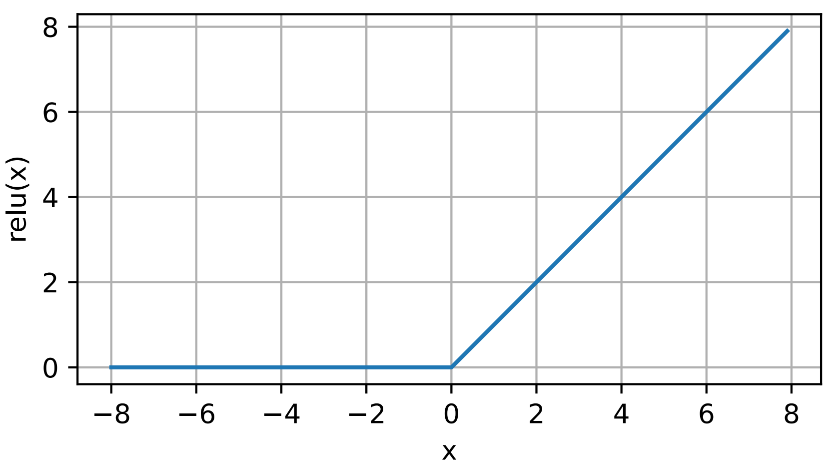
[1]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

MLP : for non-Linear calculation

FC: Fully Connected = connecting all nodes of input layer to all nodes of output layer. (one-to-one)

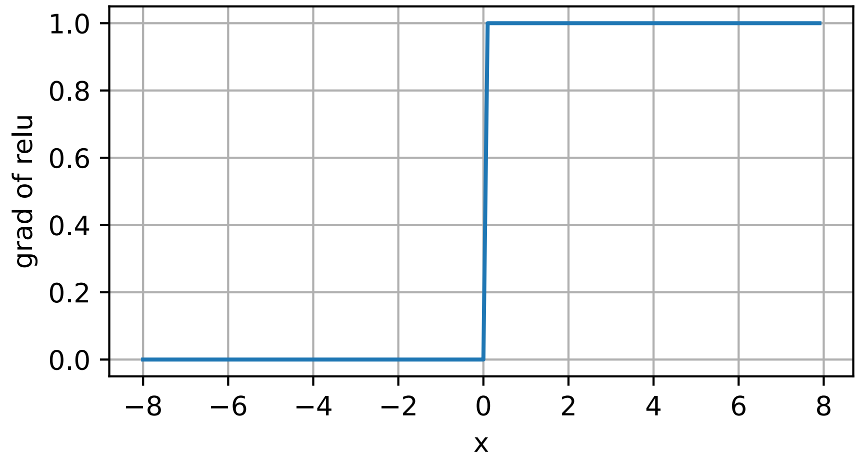
add activation function between hidden Layer and Output layer to make non-linear function

[2]:
```
#relue function : max(x,0)
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))
```

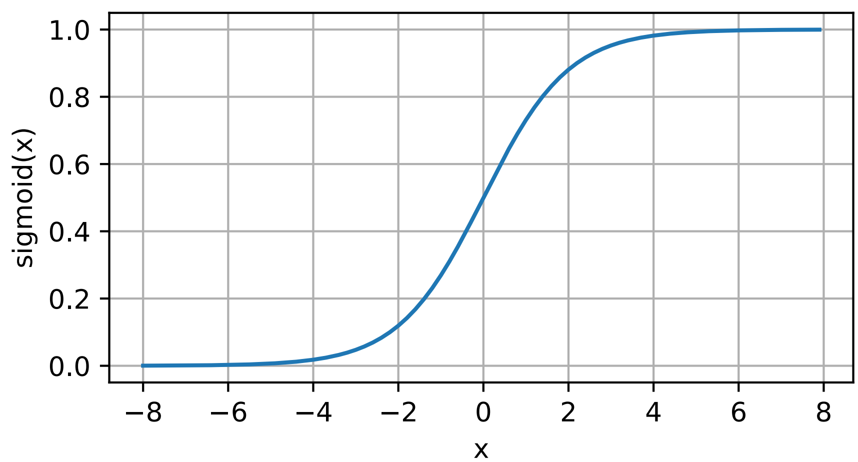


[3]:
```
# relu function
# in mathmetical approach, relu is not differentiable at x=0. However we␣
↪suppose derivative at x=0 to be 0. Because x cannot be 0.
```

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

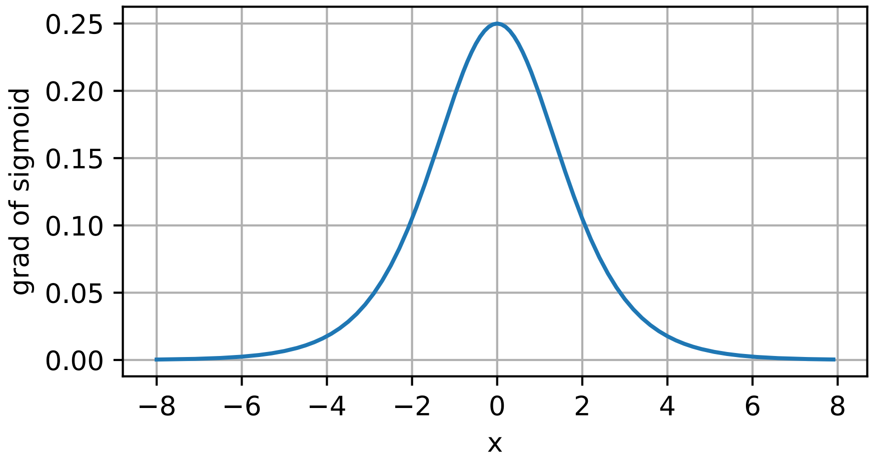


[4]:
```
#sigmoid function = squashing function
# output range from 0 to 1.
# rarely used due to vanishing gradient, only often at output layer of binary␣
↪classification. Elsewhere use Relu.
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

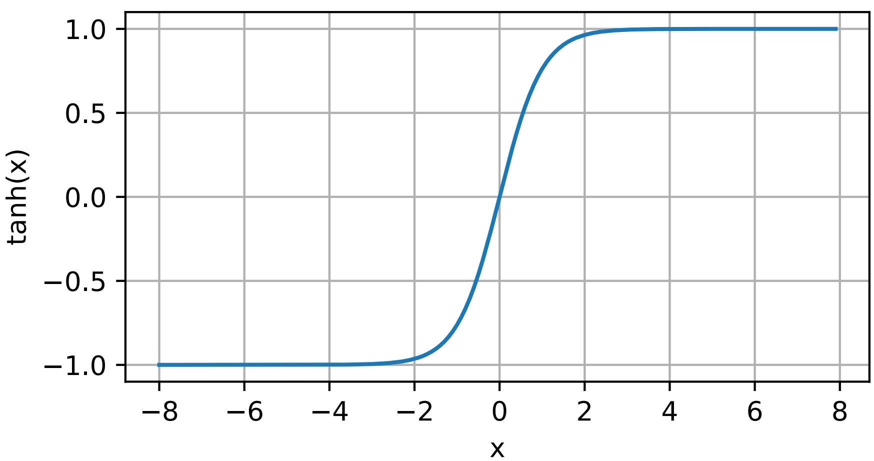


[5]:
```
# sigmoid's gradient function = sigmoid*(1-sigmoid)
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
```

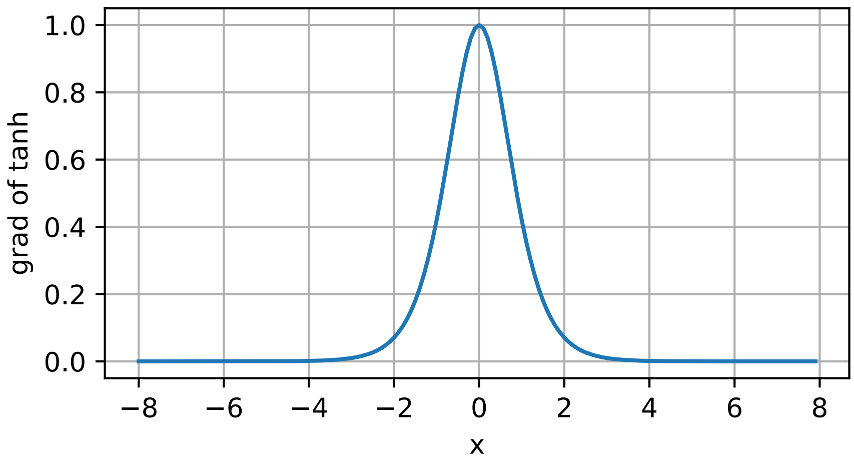
```
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5,2.5))
```



[6]:
```
# tanh function
# almost linear near x=0
# function range: (-1, 1)
# symmetric about the origin
y=torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5,2.5))
```



[7]:
```
#tanh's gradient function : 1-tanh(x)**2
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))
```



1.1 5.2.Implementation of Multilayer Preceptrons

[8]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

unit number of hidden layer used to be power of 2. (No longer effective)

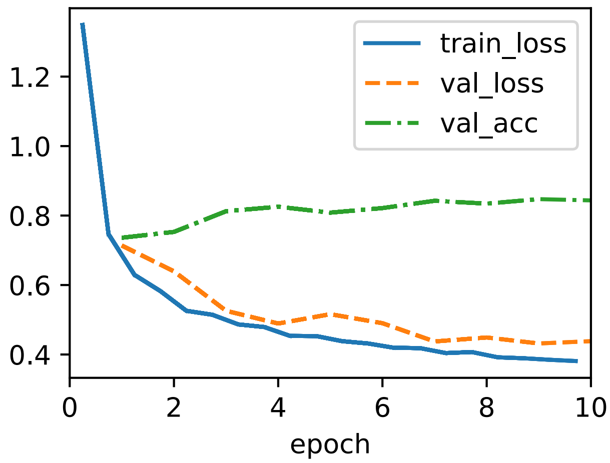
[9]:
```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

[10]:
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

[11]:
```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs)) #flatten
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2
```

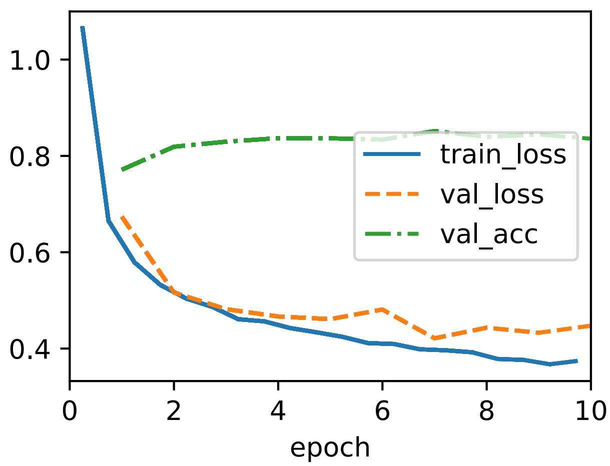
[12]:
```
# Always make "model, data, trainer" and  train the model by calling "fit".
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
```

```
trainer.fit(model, data)
```



[13]:
```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        #LazyLinear is FC layer
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))
    # forward method is not appeared because it is inherited from Module class.␣
 ↪self.net above is called for forward
```

[14]:
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



1.2 5.3.Forward Propagation, Backward Propagation, and Computational Graphs

Forward Propagation

objective function(J) = L(loss function) + s(regularization term: to prevent overfit)

Backpropagation

use chain rule, use intermediate variables' value stashed when forwarding

1. objective function(J) partial derivative by loss function(L) = 1 2. J partial derivative by output(O) 3. s partial derivative by W 4. using the result of 2, J partial derivative by W

Q. What is regularization term?

[ ]:

```

```