

## ✓ COSE474-2024F: Deep Learning

### ✓ 0.1 Installation

```
pip install d2l==1.0.3
```



```
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages:
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10,
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10,
Requirement already satisfied: prompt-toolkit!=3.0.0,!3.0.1,<3.1.0,>=2.0.0 in /usr/lo
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (fr
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from r
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packag
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-pa
```

```
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (cffi)
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages
```

## ✓ 2.1 Data Manipulation

```
import torch
```

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

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

```
x.numel()
```

```
⇒ 12
```

```
x.shape
```

```
⇒ torch.Size([12])
```

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

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

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

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

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

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

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

```
⇒ tensor([[ 1.2023,  0.1389,  0.6277,  0.6239],  
          [-0.6917, -1.5976,  0.0926, -0.8989],  
          [ 0.9984, -0.6122, -0.5318,  0.3574]])
```

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

```
⇒ tensor([[2, 1, 4, 3],  
          [1, 2, 3, 4],  
          [4, 3, 2, 1]])
```

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

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

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

```
X
```

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

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

```
X
```

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

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

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

```
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
```

```
⇒ (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.]))
```

```
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)
```

```
⇒ (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.])))
```

```
X == Y
```

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

```
X.sum()
```

```
⇒ tensor(66.)
```

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

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

```
a + b
```

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

```
before = id(Y)
Y = Y + X
id(Y) == before
```

⇒ False

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

⇒ id(Z): 135136493353088  
id(Z): 135136493353088

```
before = id(X)
X += Y
id(X) == before
```

⇒ True

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

⇒ (numpy.ndarray, torch.Tensor)

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

⇒ (tensor([3.5000]), 3.5, 3.5, 3)

## ✓ 2.2 Data Preprocessing

```
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'')
```

```
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
```

```
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
```

```
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
```

```
import torch
```

```
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y
```

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

## ✓ 2.3 Linear Algebra

```
import torch
```

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)
```

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

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

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

```
⇒ tensor([0, 1, 2])
```

```
x[2]
```

```
⇒ tensor(2)
```

```
len(x)
```

```
⇒ 3
```

```
x.shape
```

```
⇒ torch.Size([3])
```

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

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

```
A.T
```

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

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

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

```
torch.arange(24).reshape(2, 3, 4)
```

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

```
[[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]])
```

```
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
```

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

A \* B

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

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

```
⇒ (tensor([[[ 2,  3,  4,  5],
             [ 6,  7,  8,  9],
             [10, 11, 12, 13]],

           [[14, 15, 16, 17],
            [18, 19, 20, 21],
            [22, 23, 24, 25]]]),
   torch.Size([2, 3, 4]))
```

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

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

A.shape, A.sum()

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

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

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

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

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



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

```
⇒ tensor(True)
```

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

```
⇒ (tensor(2.5000), tensor(2.5000))
```

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

```
⇒ (tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

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

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

```
A / sum_A
```

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

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

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

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

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

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

```
⇒ tensor(3.)
```

```
A.shape, x.shape, torch.mv(A, x), A@x
```

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

```
B = torch.ones(3, 4)  
torch.mm(A, B), A@B
```

```
⇒ tensor([[ 3.,  3.,  3.,  3.],
          [12., 12., 12., 12.])),
        tensor([[ 3.,  3.,  3.,  3.],
          [12., 12., 12., 12.])))
```

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

```
⇒ tensor(5.)
```

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

```
⇒ tensor(7.)
```

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

```
⇒ tensor(6.)
```

## ✓ 2.5 Automatic Differentiation

```
import torch
```

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

```
⇒ tensor([0., 1., 2., 3.])
```

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

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

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

```
y.backward()
x.grad
```

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

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

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

```
x.grad.zero_() # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

⇒ tensor([1., 1., 1., 1.])

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

⇒ tensor([0., 2., 4., 6.])

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

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

⇒ tensor([True, True, True, True])


```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

⇒ tensor([True, True, True, True])

```
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
```

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

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


 tensor(True)

## ✓ 3.1 Linear Regression


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

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

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

 '0.11730 sec'

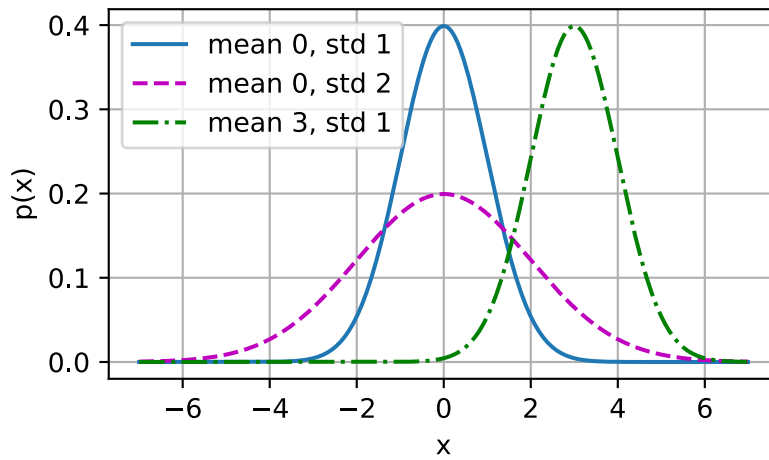
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

 '0.00262 sec'

```
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)
```

```
# 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])
```



## ✓ 3.2 Object-Oriented Design for Implementation

```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

```
def add_to_class(Class): #@save
    """Register functions as methods in creat
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

'@save' is not an allowed annotation – allowed values include [`@param`, `@title`, `@markdown`].



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

```
a = A()
```

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

```
a.do()
```



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

```
class HyperParameters: #@save
    """The base class of hyperparameters."""
```

'@save' is not an allowed annotation – allowed values include [`@param`, `@title`, `@markdown`].



```
def save_hyperparameters(self, ignore=[]):
    raise NotImplemented
```

```
# 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
```

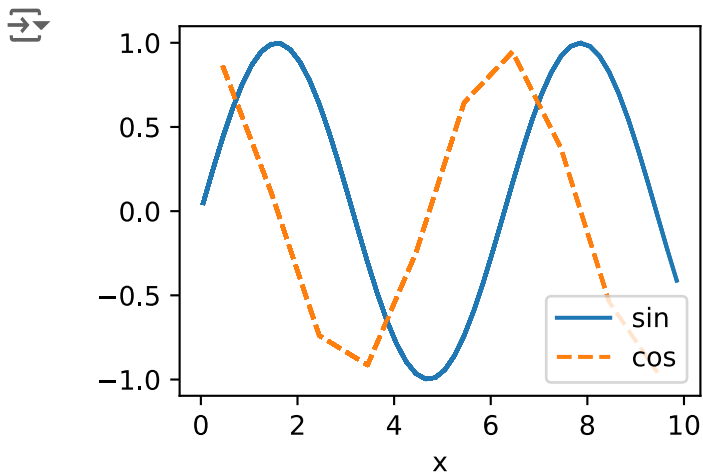
```
class ProgressBoard(d2l.HyperParameters): #@
    """The board that plots data points in an
    def __init__(self, xlabel=None, ylabel=None,
                  ylim=None, xscale='linear',
                  ls=['-', '--', '-.', ':'], c
                  fig=None, axes=None, figsize
                  self.save_hyperparameters()
```

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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
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)
```



```
class Module(nn.Module, d2l.HyperParameters):
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2):
        super().__init__()
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```

        self.save_hyperparameters()
        self.board = ProgressBoard()

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

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

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Tra
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx
            self.trainer.num_train_batche
            n = self.trainer.num_train_batche
            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()
            ('train_' if train el
            every_n=int(n))

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

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

    def configure_optimizers(self):
        raise NotImplementedError

```

```

class DataModule(d2l.HyperParameters): #@sav
    """The base class of data."""
    def __init__(self, root='../data', num_wo
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

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

```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

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

    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
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



## ✓ 3.4 Linear Regression Implementation from Scratch

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

```
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))
        self.b = torch.zeros(1, requires_grad=True)
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].





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

'@save' is not an allowed annotation – allowed values include [[@param](#), [@title](#), [@markdown](#)].



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

'@save' is not an allowed annotation – allowed values include [[@param](#), [@title](#), [@markdown](#)].



```
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_()
```

'@save' is not an allowed annotation – allowed values include [[@param](#), [@title](#), [@markdown](#)].



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

'@save' is not an allowed annotation – allowed values include [[@param](#), [@title](#), [@markdown](#)].



```
@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,
            self.optim.zero_grad()
            with torch.no_grad():
                loss.backward()
                if self.gradient_clip_val > 0: #
                    self.clip_gradients(self.grad
                    self.optim.step()
                self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
```

'@save' is not an allowed annotation – allowed values include [[@param](#), [@title](#), [@markdown](#)].



'@save' is not an allowed annotation – allowed values include [[@param](#), [@title](#), [@markdown](#)].



```

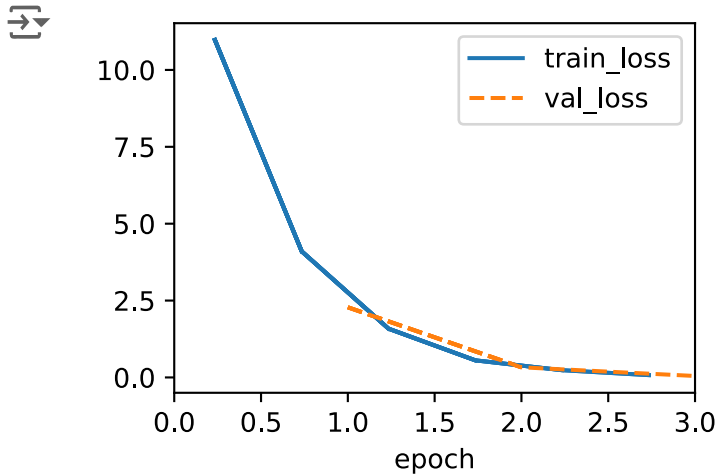
with torch.no_grad():
    self.model.validation_step(self.p
self.val_batch_idx += 1

```

```

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)

```



```

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([ 0.0958, -0.1693])
error in estimating b: tensor([0.2455])

```

## 4.1 softmax Regression

## ✓ 4.2 The Image Classification Dataset

```

%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()

```

```

class FashionMNIST(d2l.DataModule): #@save
    """The Fashion-MNIST dataset."""

```

'@save' is not an allowed annotation –  
allowed values include [@param, @title,



```
def __init__(self, batch_size=64, resize=
    super().__init__()
    self.save_hyperparameters()
    trans = transforms.Compose([transform
                                transform
    self.train = torchvision.datasets.Fas
        root=self.root, train=True, trans
    self.val = torchvision.datasets.Fashi
        root=self.root, train=False, tran
```

@markdown].

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

➞ (60000, 10000)

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

➞ torch.Size([1, 32, 32])

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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
@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,
                                        num_wo
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

➞ /usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: warnings.warn(\_create\_warning\_msg(torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64

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

➞ '13.94 sec'

```
def show_images(imgs, num_rows, num_cols, tit
    """Plot a list of images."""
    raise NotImplementedError
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].

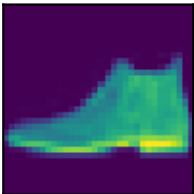


```
@d2l.add_to_class(FashionMNIST) #@save
def visualize(self, batch, nrows=1, ncols=8,
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    d2l.show_images(X.squeeze(1), nrows, ncol
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```

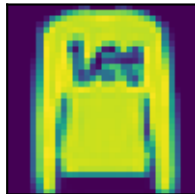
'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



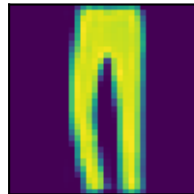
ankle boot



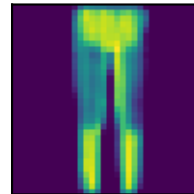
pullover



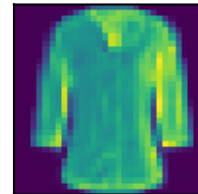
trouser



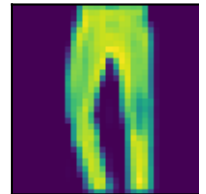
trouser



shirt



trouser



## ✓ 4.3 The Base Classification Model

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

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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
@d2l.add_to_class(Classifier) #@save
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the number of correct predicti
    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(t
    return compare.mean() if averaged else co
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



## ✓ 4.4 Softmax Regression Implementation from Scratch

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

```
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)
```

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

```
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
```

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

```
⇒ (tensor([[0.2282, 0.1459, 0.1989, 0.2840, 0.1430],
           [0.1707, 0.2255, 0.2369, 0.2389, 0.1280]]),
    tensor([1., 1.])))
```

```
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]
```

```
@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)
```

```
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]
```

⇒ tensor([0.1000, 0.5000])

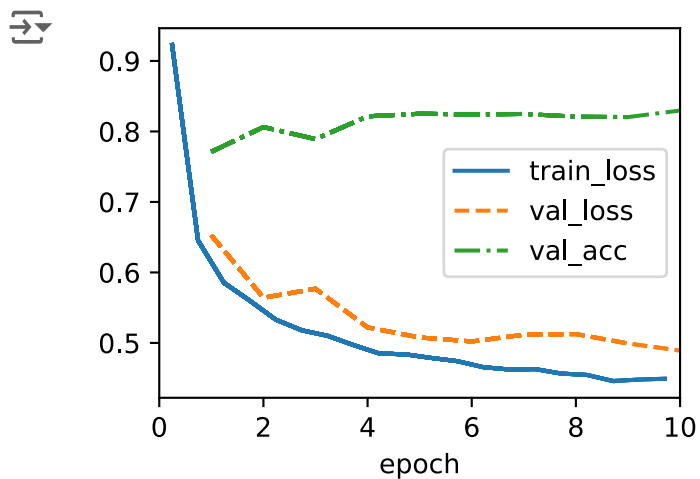
```
def cross_entropy(y_hat, y):  
    return -torch.log(y_hat[list(range(len(y_hat)))], y).mean()
```

cross\_entropy(y\_hat, y)

⇒ tensor(1.4979)

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

```
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)
```



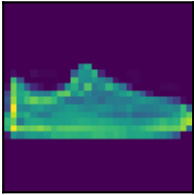
```
X, y = next(iter(data.val_dataloader()))  
preds = model(X).argmax(axis=1)  
preds.shape
```

⇒ torch.Size([256])

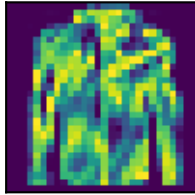
```
wrong = preds.type(y.dtype) != y  
X, y, preds = X[wrong], y[wrong], preds[wrong]  
labels = [a+'\n'+b for a, b in zip(  
    data.text_labels(y), data.text_labels(preds))]  
data.visualize([X, y], labels=labels)
```



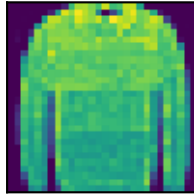
sneaker  
sandal



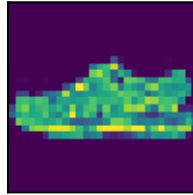
coat  
pullover



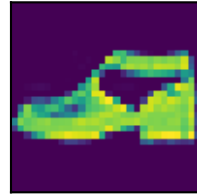
pullover  
t-shirt



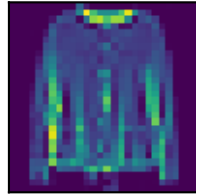
sandal  
sneaker



ankle boot  
sneaker



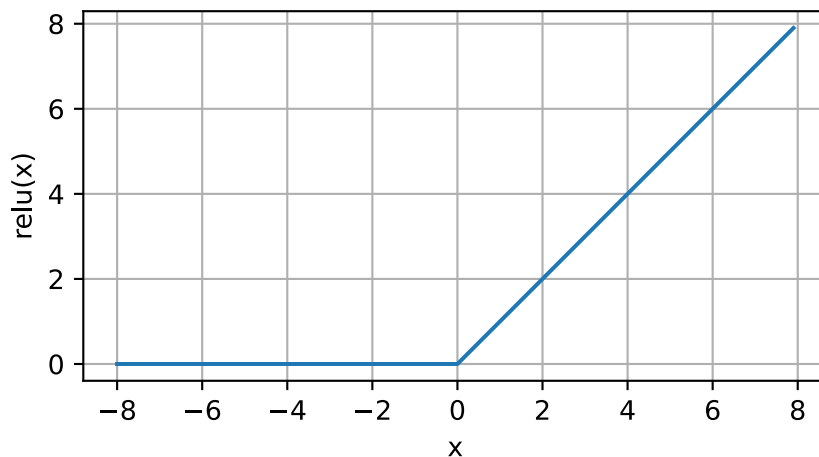
coat  
pullover



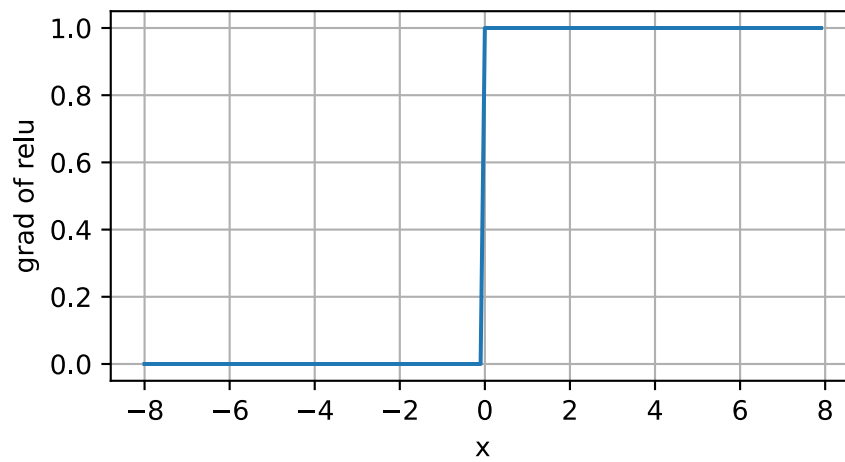
## ✓ 5.1 Multilayer Perceptrons

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

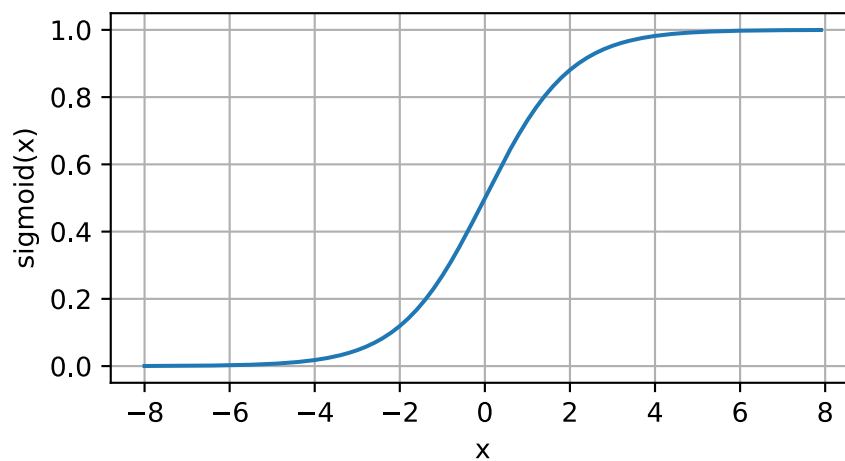
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



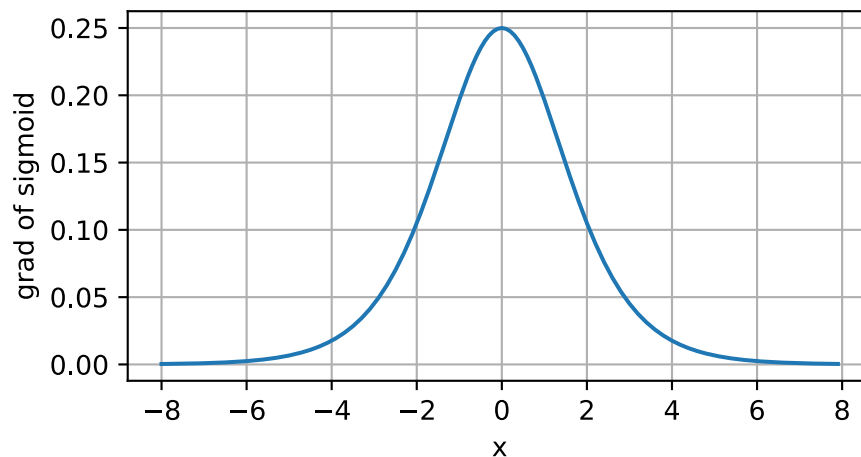
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



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

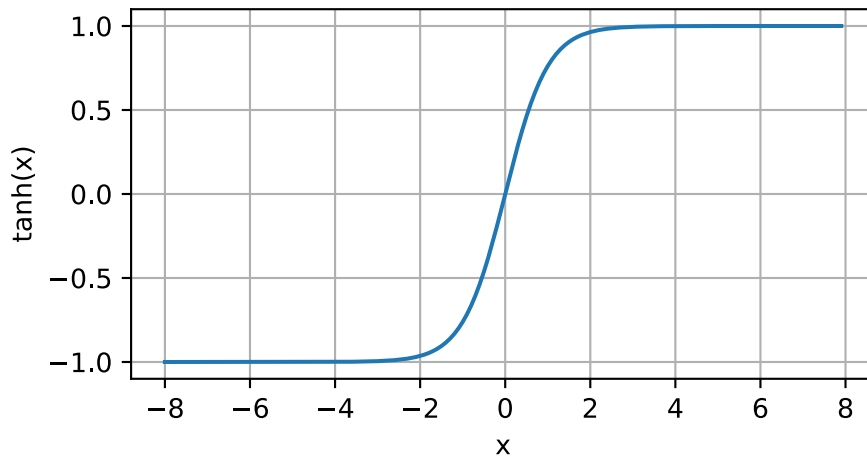


```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

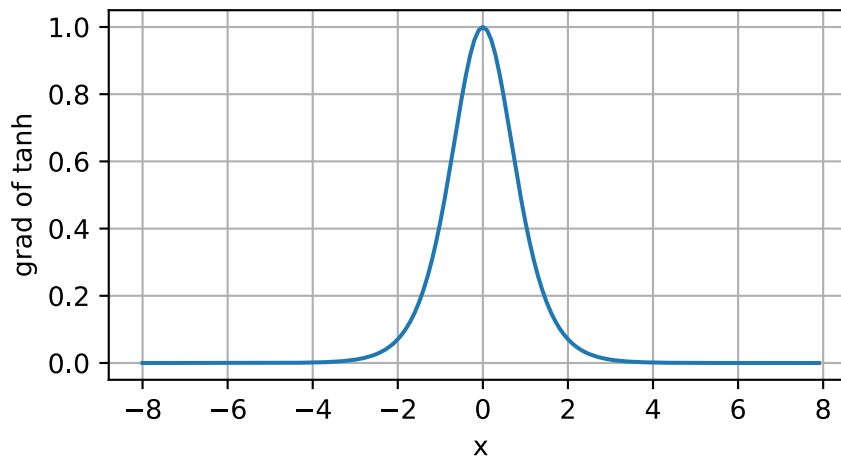




```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



```
# 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))
```



## ✓ 5.2 Implementation of Multilayer Perceptrons

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

```
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))

```

```

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

```

```

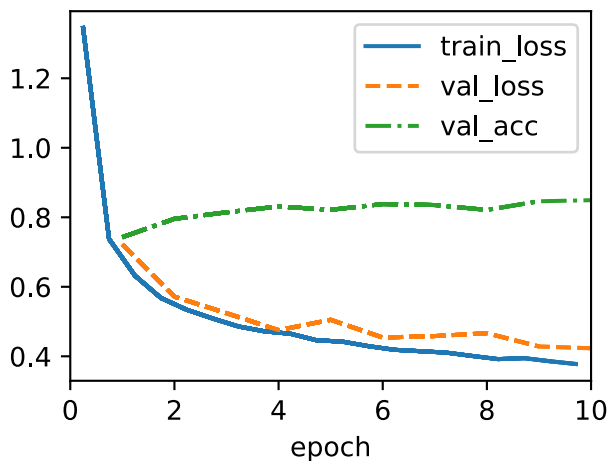
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2

```

```

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)

```



```

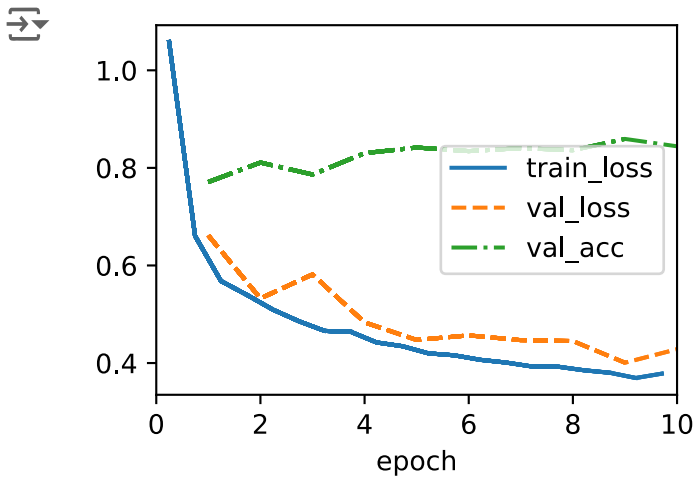
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                   nn.ReLU(), nn.LazyLinear(num_outputs))

```

```

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

```



## 5.3 Forward Propagation, Backward Propagation, and Computational Graphs

### ✓ Discussions and exercises

### ✓ 2.1 Discussion

#### 2.1.1. Getting Started

- The section explains what a tensor is and its dimensional classifications (vector, matrix, and higher-order tensors).

#### 2.1.4. Broadcasting

- Broadcasting allows elementwise operations on tensors of different shapes by expanding them to a common shape.

#### 2.1.5. Saving Memory

- In-place operations to minimize memory usage and prevent potential memory leaks when multiple variables reference the same data are crucial.

### ✓ 2.1 Exercises

1.

```
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)
x == y, x > y, x < y
```

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

2.

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

```
⇒ (tensor([[[ 0],
            [ 1],
            [ 2],
            [ 3]],

          [[ 4],
            [ 5],
            [ 6],
            [ 7]],

          [[ 8],
            [ 9],
            [10],
            [11]]]),
   tensor([[[ 0,  1,  2],
            [ 3,  4,  5],
            [ 6,  7,  8],
            [ 9, 10, 11]]]),
   tensor([[[ 0,  1,  2],
            [ 4,  5,  6],
            [ 8,  9, 10],
            [12, 13, 14]],

          [[ 4,  5,  6],
            [ 8,  9, 10],
            [12, 13, 14],
            [16, 17, 18]],

          [[ 8,  9, 10],
            [12, 13, 14],
            [16, 17, 18],
            [20, 21, 22]]]))
```

## ✓ 2.2 Discussion

### 2.2.2. Data Preparation

- The importance of distinguishing between input features and target values in supervised learning is discussed, along with methods for selecting these columns using pandas.
- Missing values are identified and handled through imputation (replacing with estimates) and deletion (removing affected rows or columns).

## ✓ 2.2 Exercises

1.

```
import pandas as pd
```

```
link = "https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data"
columns = ["Sex", "Length", "Diameter", "Height", "Whole_weight",
           "Shucked_weight", "Viscera_weight", "Shell_weight", "Rings"]
abalone = pd.read_csv(link, names=columns)
abalone.isna().mean()
```




	0
<b>Sex</b>	0.0
<b>Length</b>	0.0
<b>Diameter</b>	0.0
<b>Height</b>	0.0
<b>Whole_weight</b>	0.0
<b>Shucked_weight</b>	0.0
<b>Viscera_weight</b>	0.0
<b>Shell_weight</b>	0.0
<b>Rings</b>	0.0

**dtype:** float64


```
numerical = round(abalone.select_dtypes(include=["number"]).shape[1] / abalone.shape[1], 3)
categorical = round(abalone.select_dtypes(include=["object"]).shape[1] / abalone.shape[1], 3)
```



numerical, categorical

 (0.889, 0.111)

2.

```
abalone[["Sex", "Diameter", "Whole_weight", "Rings"]].head()
```



	Sex	Diameter	Whole_weight	Rings	
0	M	0.365	0.5140	15	
1	M	0.265	0.2255	7	
2	F	0.420	0.6770	9	
3	M	0.365	0.5160	10	
4	I	0.255	0.2050	7	

3.

Start coding or [generate](#) with AI.

4. For large numbers of categories:

- Infrequent categories can be combined into one single group to reduce dimensionality.
- Encodings such as One-Hot, Target Encoding, and Embeddings can be used depending on set size.
- Unique labels can also be excluded if they do not provide any generalizeable information.

5.

## ✓ 2.3 Discussion

- Introduces essential linear algebra concepts needed for building machine learning models. Concepts range from scalar arithmetic to matrix multiplication, all implemented using tensor operations in libraries such as PyTorch and TensorFlow.

### 2.3.4. Tensors

- Higher-order tensors extend vectors and matrices by increasing the number of axes.

- Tensors are used for complex data such as images, where each pixel has multiple channels (color intensities).

### 2.3.7. Non-Reduction Sum

- Reducing a tensor without collapsing its axes can be useful, for example, when using broadcasting to scale tensors.
- Functions like cumulative sum are also supported, which accumulate sums along a specific axis without reducing the tensor's shape.

## ✓ 2.3 Exercises

1.

```
import torch

A = torch.arange(8, dtype=torch.float32).reshape(2, 4)
(A.T).T == A
```

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

2.

```
B = A.clone()
A.T + B.T == (A + B).T
```

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

3.

```
A = torch.arange(9, dtype=torch.float32).reshape(3, 3)
A + A.T == (A + A.T).T
```

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

4. If the tensor X has shaper (2, 3, 4), the output of len(X) is 2, as it returns the first dimension of the tensor, in this case being 2.

```
X = torch.arange(24).reshape(2, 3, 4)
len(X)
```

⇒ 2

5. len(x) will always correspond to the first axis (0).

6.

```
A = torch.arange(8, dtype=torch.float32).reshape(2, 4)
A / A.sum(axis=1)
```

⇒

```
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-15-269d379ed784> in <cell line: 2>()
      1 A = torch.arange(8, dtype=torch.float32).reshape(2, 4)
----> 2 A / A.sum(axis=1)
```

**RuntimeError:** The size of tensor a (4) must match the size of tensor b (2) at non-singleton dimension 1

Next steps: [Explain error](#)

7. In locations with grid-like structures like Manhattan, the Manhattan distance can be used to measure the shortest path where only right angles are followed. Moving diagonally would be impossible due to the grid-like structure of the city as there are no such paths.

8.

```
X = torch.arange(24).reshape(2, 3, 4)
X , X.sum(axis=0), X.sum(axis=1), X.sum(axis=2)
```

⇒

```
(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]]]),
 tensor([[12, 14, 16, 18],
         [20, 22, 24, 26],
```



```

        [28, 30, 32, 34]]),
tensor([[12, 15, 18, 21],
        [48, 51, 54, 57]]),
tensor([[ 6, 22, 38],
        [54, 70, 86]]))

```

9.

```

X = torch.arange(24, dtype=torch.float32).reshape(2, 3, 4)
torch.linalg.norm(X)

```

```

→ tensor(65.7571)

```

10. Computing  $(AB)C$  is much faster than  $A(BC)$ , as the matrix  $AB$  has size  $2^{10} \times 2^5$ , while the matrix  $BC$  has size  $2^{16} \times 2^{14}$ .

12.

```

torch.manual_seed(42)
A = torch.randn(100, 200, dtype=torch.float64)
B = torch.randn(100, 200, dtype=torch.float64)
C = torch.randn(100, 200, dtype=torch.float64)
torch.stack([A,B,C]).shape, torch.stack([A,B,C])[1] == B

```

```

→ (torch.Size([3, 100, 200]),
   tensor([[True, True, True, ..., True, True, True],
          [True, True, True, ..., True, True, True],
          [True, True, True, ..., True, True, True],
          ...,
          [True, True, True, ..., True, True, True],
          [True, True, True, ..., True, True, True],
          [True, True, True, ..., True, True, True]]))

```

## ✓ 2.5 Discussion

- Automatic differentiation (autograd) simplifies the process of calculating derivatives, which are critical in training deep learning models.
- Frameworks like PyTorch track computational steps and calculate gradients using the chain rule via backpropagation.

### 2.5.1. A Simple Function

- Demonstrates how to compute gradients of a simple function  $f(x)$  using PyTorch.

### 2.5.2. Backward for Non-Scalar Variables

- Explains that for non-scalar outputs, the gradient is a Jacobian matrix.
- However, in deep learning, we often sum gradients over a batch, resulting in a vector instead of a matrix.

#### 2.5.3. Detaching Computation

- In some cases, it is necessary to detach a computation from the computational graph to prevent gradients from propagating through certain operations.

#### 2.5.4. Gradients and Python Control Flow

- Automatic differentiation works even when the computation involves dynamic control flow.
- Regardless of how a graph is constructed, gradients can still be computed.

## ✓ 2.5 Exercises

1. The second derivative is more expensive because it requires calculating the derivative of the first derivative, which involves additional operations and storage for intermediate results. This means more calls to `backward()` and handling larger computation graphs, which increase the complexity.

2.

```
import torch

x = torch.arange(4.0)
x.requires_grad_(True)
x.grad # The gradient is None by default
y = 2 * torch.dot(x, x)
y.backward()
y.backward()
x.grad
```



```
-----  
RuntimeError                                Traceback (most recent call last)  
<ipython-input-19-8aac0592d564> in <cell line: 8>()  
      6 y = 2 * torch.dot(x, x)  
      7 y.backward()  
----> 8 y.backward()  
      9 x.grad
```

2 frames

```
/usr/local/lib/python3.10/dist-packages/torch/autograd/graph.py in  
_engine_run_backward(t_outputs, *args, **kwargs)  
    767     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)  
    768     try:  
--> 769     return Variable._execution_engine.run_backward( # Calls into the C++  
engine to run the backward pass  
    770         t_outputs, *args, **kwargs  
    771     ) # Calls into the C++ engine to run the backward pass
```

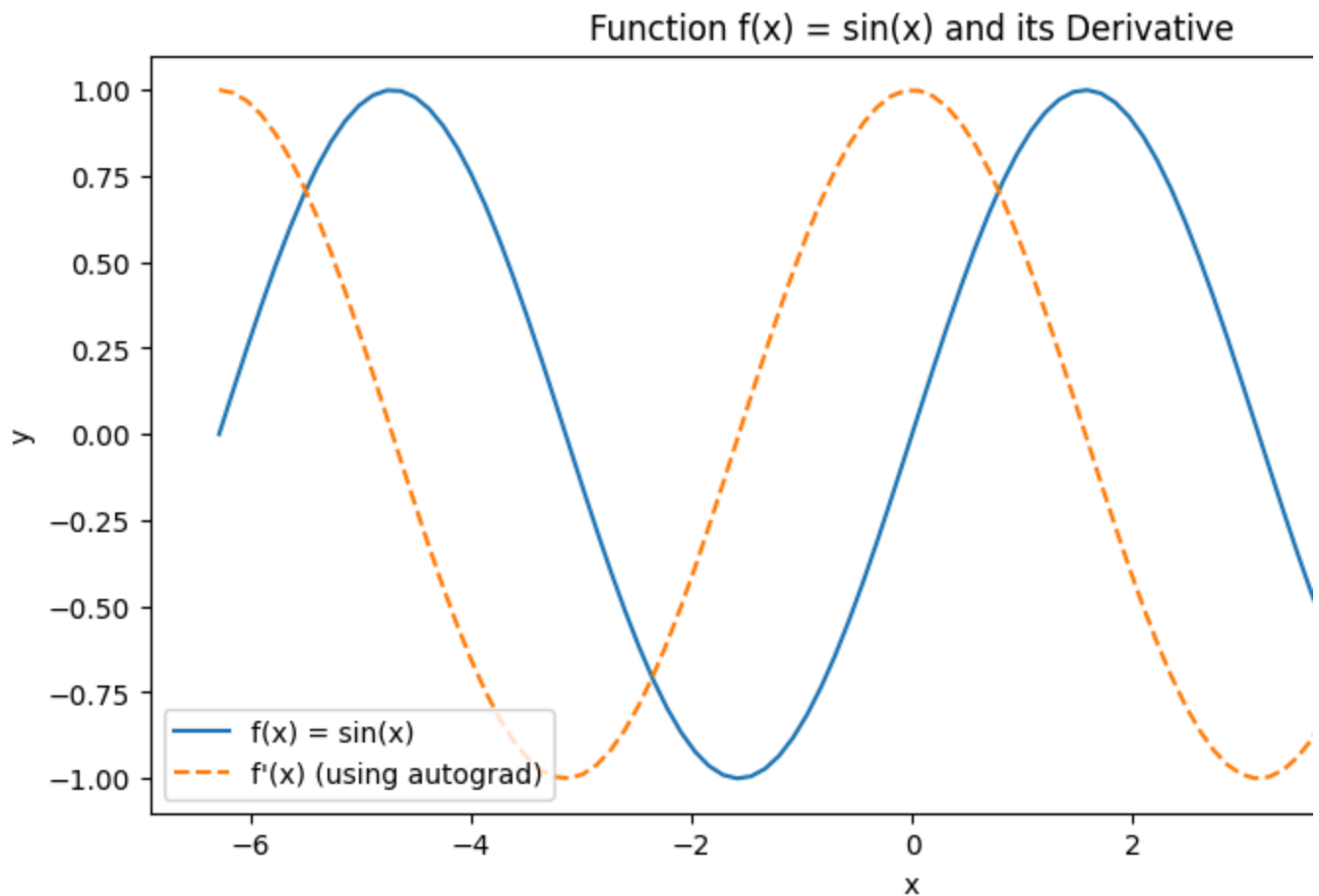
**RuntimeError:** Trying to backward through the graph a second time (or directly access saved tensors after they have already been freed). Saved intermediate values of the graph are freed when you call `.backward()` or `autograd.grad()`. Specify `retain_graph=True` if you need to backward through the graph a second time or if you need to access saved tensors after calling backward.

Next steps: [Explain error](#)

4.

```
import numpy as np  
import matplotlib.pyplot as plt  
  
# Define the function f(x) = sin(x)  
x = torch.linspace(-2 * torch.pi, 2 * torch.pi, 100, requires_grad=True)  
f_x = torch.sin(x)  
  
# Compute the derivative using PyTorch's autograd  
f_x.backward(torch.ones_like(x))  
f_prime = x.grad  
  
# Convert to NumPy for plotting  
x_np = x.detach().numpy()  
f_x_np = f_x.detach().numpy()  
f_prime_np = f_prime.detach().numpy()  
  
# Plot the function and its derivative  
plt.figure(figsize=(10, 5))  
plt.plot(x_np, f_x_np, label="f(x) = sin(x)")  
plt.plot(x_np, f_prime_np, label="f'(x) (using autograd)", linestyle='--')
```

```
plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.title("Function f(x) = sin(x) and its Derivative")
plt.show()
```



5.

```
# Create a range of x values for the plot
x_vals = torch.linspace(0.5, 5, 100, requires_grad=True)

# Define the function f(x) in one line
f_x_vals = (torch.log(x_vals ** 2)) * torch.sin(x_vals) + x_vals.pow(-1)

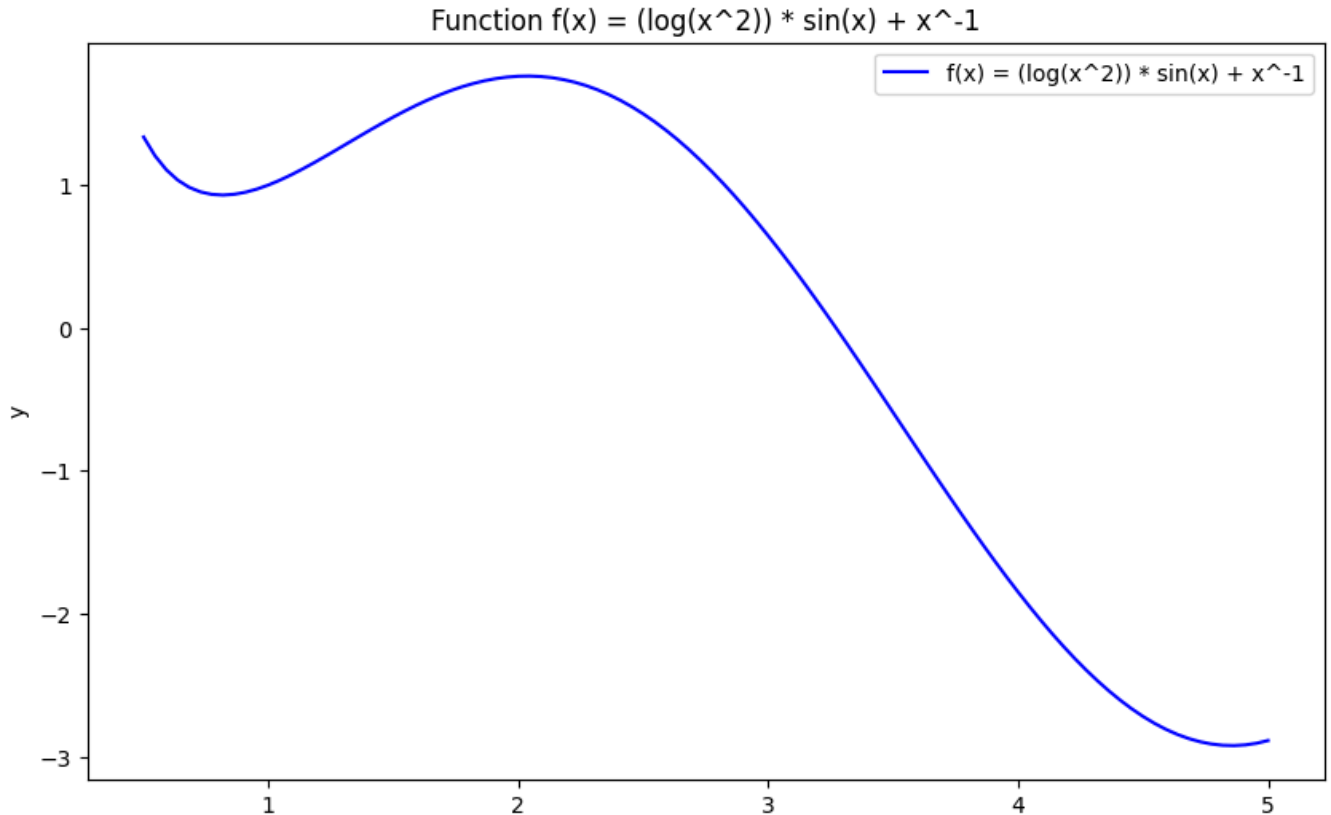
# Convert tensors to NumPy arrays for plotting
x_vals_np = x_vals.detach().numpy()
f_x_vals_np = f_x_vals.detach().numpy()

# Plot the function f(x)
plt.figure(figsize=(10, 6))
plt.plot(x_vals_np, f_x_vals_np, label="f(x) = (log(x^2)) * sin(x) + x^-1", color="blue")

# Add labels and title
```

```
plt.xlabel("x")
plt.ylabel("y")
plt.title("Function  $f(x) = (\log(x^2)) * \sin(x) + x^{-1}$ ")
plt.legend()

# Show the plot
plt.show()
```



6.

```
# Create a range of x values for the plot
x_vals = torch.linspace(0.5, 5, 100, requires_grad=True)

# Define the function  $f(x) = (\log(x^2)) * \sin(x) + x^{-1}$ 
f_x_vals = (torch.log(x_vals ** 2)) * torch.sin(x_vals) + x_vals.pow(-1)

# Compute the derivative of f with respect to x
f_x_vals.backward(torch.ones_like(x_vals))

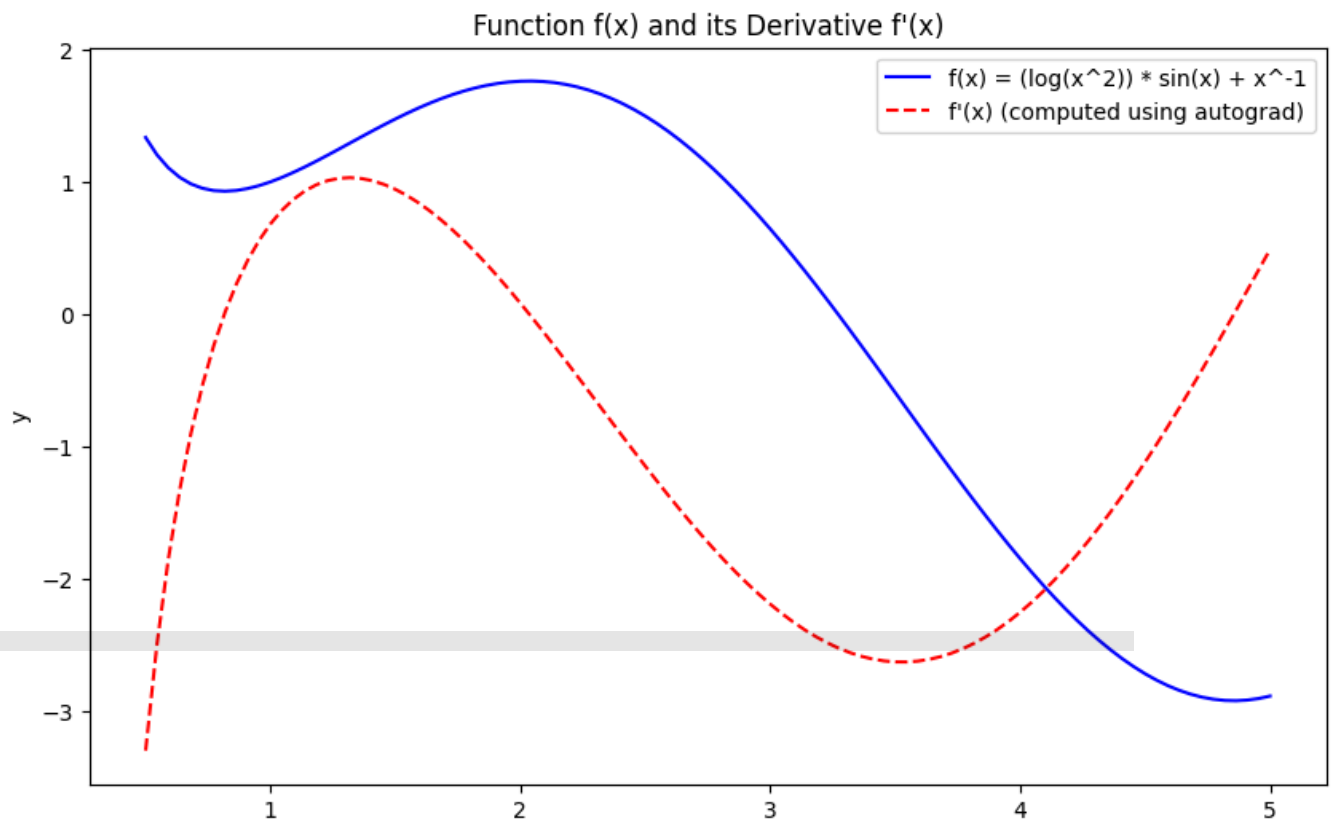
# Get the derivative (gradient)
f_prime_vals = x_vals.grad

# Convert tensors to NumPy arrays for plotting
x_vals_np = x_vals.detach().numpy()
f_x_vals_np = f_x_vals.detach().numpy()
f_prime_vals_np = f_prime_vals.detach().numpy()
```

```
# Plot the function and its derivative
plt.figure(figsize=(10, 6))
plt.plot(x_vals_np, f_x_vals_np, label="f(x) = (log(x^2)) * sin(x) + x^-1", color="blue")
plt.plot(x_vals_np, f_prime_vals_np, label="f'(x) (computed using autograd)", color="red", linestyle='dashed')

# Add labels and title
plt.xlabel("x")
plt.ylabel("y")
plt.title("Function f(x) and its Derivative f'(x)")
plt.legend()

# Show the plot
plt.show()
```



7.

Start coding or [generate](#) with AI.

8. Forward differentiation works best when there are few inputs and many outputs, as it efficiently computes derivatives from inputs to outputs.

Backward differentiation works best when there are many inputs and few outputs. This happens because backward differentiation computes the gradients with respect to the inputs in one pass after the forward computation, reducing the number of operations and memory usage.

## ✓ 3.1 Discussion

### 3.1.1.4 Minibatch Stochastic Gradient Descent (SGD):

- Minibatch SGD provides a practical way to optimize model parameters by using small subsets of data to update weights iteratively. This balances computational efficiency with statistical accuracy and is crucial for training large models.

### 3.1.2 Vectorization for Speed:

- Vectorization dramatically speeds up calculations by leveraging linear algebra operations over individual loops in Python. This optimization is critical when working with large datasets and is more efficient in memory and computation.

### 3.1.4 Linear Regression as a Neural Network:

- Linear regression can be viewed as a single-layer neural network, where each feature is connected directly to the output. This highlights the close relationship between simple linear models and neural networks.

## ✓ 3.1 Exercises

1. The function to minimize is:

$$f(b) = \sum_{i=1}^n (x_i - b)^2$$

The derivative with respect to  $b$ :

$$f'(b) = \sum_{i=1}^n -2(x_i - b)$$

Setting the derivative to zero and solving for  $b$  gives:

$$0 = \sum_{i=1}^n -2(x_i - b)$$

This leads to:

$$b = \frac{1}{n} \sum_{i=1}^n x_i$$

So, the value of  $b$  that minimizes the sum of squared differences is the mean of the  $x_i$ .

This relates to the normal distribution because minimizing the sum of squared differences helps find the mean, which is the most likely value for a normal distribution.

If the focus is changed to the sum of absolute differences instead, the median of the  $x_i$  must be found. The median minimizes the sum of absolute differences and is more reliable when there are outliers in the data.

2. An affine function combines a linear transformation with a translation, often looking like:

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b$$

, where  $\mathbf{x}$  is the input vector,  $\mathbf{w}$  is the weight vector, and  $b$  is the bias.

This affine function can be seen as a linear function when using an augmented vector  $(\mathbf{x}, 1)$ . By defining a new weight vector  $\mathbf{w}' = (\mathbf{w}, b)$  and an augmented input vector  $\mathbf{x}' = (\mathbf{x}, 1)$ . Then the affine function can be expressed as:

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b = \mathbf{x}'^\top \mathbf{w}'$$

By starting with the function  $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b$ ,  $\mathbf{x}' = (\mathbf{x}, 1)$  and  $\mathbf{w}' = (\mathbf{w}, b)$  can be defined. The dot product  $\mathbf{x}'^\top \mathbf{w}'$  results in  $\mathbf{x}^\top \mathbf{w} + 1 \cdot b$ , which matches the original function.

Thus, the function  $\mathbf{x}^\top \mathbf{w} + b$  is equivalent to the linear function  $\mathbf{x}'^\top \mathbf{w}'$  when using the augmented vector  $(\mathbf{x}, 1)$ . This equivalence helps simplify notation and implementation.

3. To find quadratic functions of  $\mathbf{x}$ , like:

$$f(\mathbf{x}) = b + \sum_i w_i x_i + \sum_{j \leq i} w_{ij} x_i x_j$$

in a deep network, the following architecture can be used:

- Input Layer: The input to the network is the vector  $\mathbf{x} = [x_1, x_2, \dots, x_n]$ , which contains the features for which the quadratic function is to be learned.
- First Fully Connected Layer: This layer performs a linear transformation on the input vector, where the weights represent the  $w_i$  terms in the quadratic function. The output of this layer is

$$b + \sum_i w_i x_i$$

Here  $w_i$  are the weights of the layer and  $b$  is the bias term.

- Element-wise Multiplication Layer: To capture the quadratic terms  $x_i x_j$ , perform an element-wise multiplication of the input vector  $\mathbf{x}$  with itself. This operation results in a vector  $\mathbf{x} \odot \mathbf{x}$ . If cross terms  $x_i x_j$  for  $i \neq j$  are needed, a more complex approach like pairwise interaction between elements of  $\mathbf{x}$  can be applied.
- Second Fully Connected Layer: This layer processes the output of the element-wise multiplication (or pairwise interaction), where the weights now represent the  $w_{ij}$  terms in the



quadratic function. The output is:

$$\sum_{j \leq i} w_{ij} x_i x_j$$

- Summation Layer: Finally, the outputs from the first fully connected layer and the second fully connected layer (quadratic terms) are summed together to produce the final result:

$$f(\mathbf{x}) = b + \sum_i w_i x_i + \sum_{j \leq i} w_{ij} x_i x_j$$

4.1 If  $\mathbf{X}^\top \mathbf{X}$  doesn't have full rank, it means that some columns in the design matrix  $\mathbf{X}$  are linearly dependent (i.e., some features are redundant or correlated). This causes problems because the matrix cannot be inverted, which means the linear regression problem does not have a unique solution. As a result, it's impossible to find one set of coefficients that minimize the error.

4.2 To fix this issue:

- Add noise: Adding a small amount of random Gaussian noise to the entries of  $\mathbf{X}$  can help make the columns of  $\mathbf{X}$  independent, which ensures  $\mathbf{X}^\top \mathbf{X}$  has full rank and can be inverted. This helps the model find a unique solution.
- Use regularization: Techniques like ridge regression or lasso regression can help by adding a penalty term, which prevents the model from focusing too much on any one feature.

4.3 If you add zero-mean Gaussian noise to the entries of  $\mathbf{X}$ , the expected value of  $\mathbf{X}^\top \mathbf{X}$  remains the same. This is because the added noise has a mean of zero, so it doesn't change the overall average of the matrix.

4.4 Stochastic Gradient Descent (SGD) can still be used even if  $\mathbf{X}^\top \mathbf{X}$  doesn't have full rank. Since SGD updates the coefficients using random subsets of the data, it doesn't require the matrix to be invertible. However, because of the linearly dependent columns, the loss surface might have multiple solutions, meaning SGD could converge to different results depending on where it starts.

5.1 The likelihood function for the data is:

$$P(\mathbf{y} | \mathbf{X}) = \prod_{i=1}^n \frac{1}{2} \exp(-|y_i - (\mathbf{x}_i^\top \mathbf{w} + b)|)$$

where  $y_i$  is the target value,  $\mathbf{x}_i$  is the input vector,  $\mathbf{w}$  is the weight vector, and  $b$  is the bias.

Taking the negative logarithm of this gives the negative log-likelihood:

$$-\log P(\mathbf{y} | \mathbf{X}) = \sum_{i=1}^n (\log 2 + |y_i - (\mathbf{x}_i^\top \mathbf{w} + b)|)$$

5.2 This problem does not have a simple solution because the absolute value in the log-likelihood makes the function non-differentiable at zero. As a result, it's not possible to find the weights  $\mathbf{w}$  and

bias  $b$  by setting the derivative equal to zero and solving.

5.3 To solve this problem, we can use a minibatch stochastic gradient descent (SGD) algorithm with the following steps:

1. Initialize the weights  $\mathbf{w}$  and bias  $b$  with random values.
2. For each minibatch of data:
  - Compute the gradient of the negative log-likelihood with respect to  $\mathbf{w}$  and  $b$ . The gradient will differ depending on whether  $y_i - (\mathbf{x}_i^\top \mathbf{w} + b)$  is positive or negative.
  - Update  $\mathbf{w}$  and  $b$  by moving in the direction of the negative gradient.
3. Repeat until the algorithm converges.

A problem may occur when  $\mathbf{w}$  and  $b$ . The gradient will differ depending on whether  $y_i - (\mathbf{x}_i^\top \mathbf{w} + b)$  is close to zero because the gradient of the absolute value function is not defined at zero. This could lead to very large updates and cause the algorithm to fail.

To address this, the following measures could be taken:

- Using regularization methods such as gradient descent with momentum or RMSProp. These techniques adjust the update step to prevent the changes from becoming too large.
- Adding a small constant to the absolute value inside the logarithm to make it differentiable, ensuring smoother updates.

6. A neural network with two linear layers, where the output of the first layer is the input to the second, won't work effectively because of the following reason:

Composition of linear layers is still linear: If you stack two linear layers without any non-linear activation in between, the overall transformation will still be equivalent to a single linear transformation. Mathematically, if the first layer is  $\mathbf{y} = \mathbf{W}_1 \mathbf{x} + b_1$  and the second layer is  $\mathbf{z} = \mathbf{W}_2 \mathbf{y} + b_2$ , then the final output is:

$$\mathbf{z} = \mathbf{W}_2(\mathbf{W}_1 \mathbf{x} + b_1) + b_2$$

This simplifies to:

$$\mathbf{z} = (\mathbf{W}_2 \mathbf{W}_1) \mathbf{x} + (\mathbf{W}_2 b_1 + b_2)$$

Which is still a linear transformation from  $\mathbf{x}$  to  $\mathbf{z}$ .

Since a neural network's strength comes from its ability to model complex, non-linear relationships, a model with only linear layers can't capture those complexities, no matter how many layers are stacked. This is why non-linear activation functions are essential. They introduce non-linearity, allowing the network to approximate more complex functions and solve a broader range of problems.

7. Estimating realistic house or stock prices using regression can be difficult because these markets are highly complex. Prices are affected by many factors, some of which are hard to measure or may not be accessible for inclusion in a regression model.

7.1. The additive Gaussian noise assumption implies that errors are symmetric and can take any value, including negative ones. However, prices (like house or stock prices) cannot be negative, making this assumption unrealistic. Fluctuations in prices are not evenly distributed; they are typically skewed, especially for financial data.

7.2. Using the logarithm of the price ( $y = \log(\text{price})$ ) is better because it transforms the data to handle multiplicative changes rather than additive ones. This models price changes more realistically, as prices often fluctuate proportionally, and this approach naturally avoids predicting negative prices.

7.3. For pennystocks (very low-priced stocks), price fluctuations are more significant in relative terms. Small changes can have a large percentage impact. Additionally, because stocks cannot trade at all possible prices due to tick sizes, this creates pricing granularity, which is more problematic for cheap stocks. In low-priced stocks, this makes price movements less smooth and predictable.

8.1. The Gaussian additive noise model may not be suitable for estimating the number of apples sold in a grocery store for several reasons:

- Negative Values: The Gaussian model allows for negative values, but the number of apples sold cannot be negative.
- Discrete vs. Continuous: The Gaussian model is a continuous distribution, while the number of apples sold is a discrete quantity.
- Large Deviations: The Gaussian model assumes that significant deviations from the mean are very unlikely. However, there can be large fluctuations in apple sales due to various factors, such as seasonal demand or promotions.

8.2 The Poisson distribution is a discrete probability distribution that describes the likelihood of a specific number of events (apples sold) occurring within a fixed time or space. The parameter  $\lambda$  represents the rate of these events. The expected value of a random variable following a Poisson distribution is  $\lambda$ . This is demonstrated by:

$$E[k] = \sum_{k=0}^{\infty} k \cdot p(k | \lambda) = \sum_{k=0}^{\infty} k \cdot \frac{\lambda^k e^{-\lambda}}{k!}$$

By the properties of the Poisson distribution, this sum equals  $\lambda$

8.3 To create a loss function for the Poisson distribution, we use the negative log-likelihood of the observed data given the predicted rates. Given observed counts  $y_1, y_2, \dots, y_n$  and predicted rates

$\lambda_1, \lambda_2, \dots, \lambda_n$ , the loss function is:

$$L(\lambda, y) = - \sum_{i=1}^n (y_i \log \lambda_i - \lambda_i)$$

Which can be simplified to:

$$L(\lambda, y) = \sum_{i=1}^n (\lambda_i - y_i \log \lambda_i)$$

This loss function measures the difference between the actual counts and the predicted rates, allowing us to fit models to count data effectively.

## ✓ 3.2 Discussion

### 3.2.1. Utilities

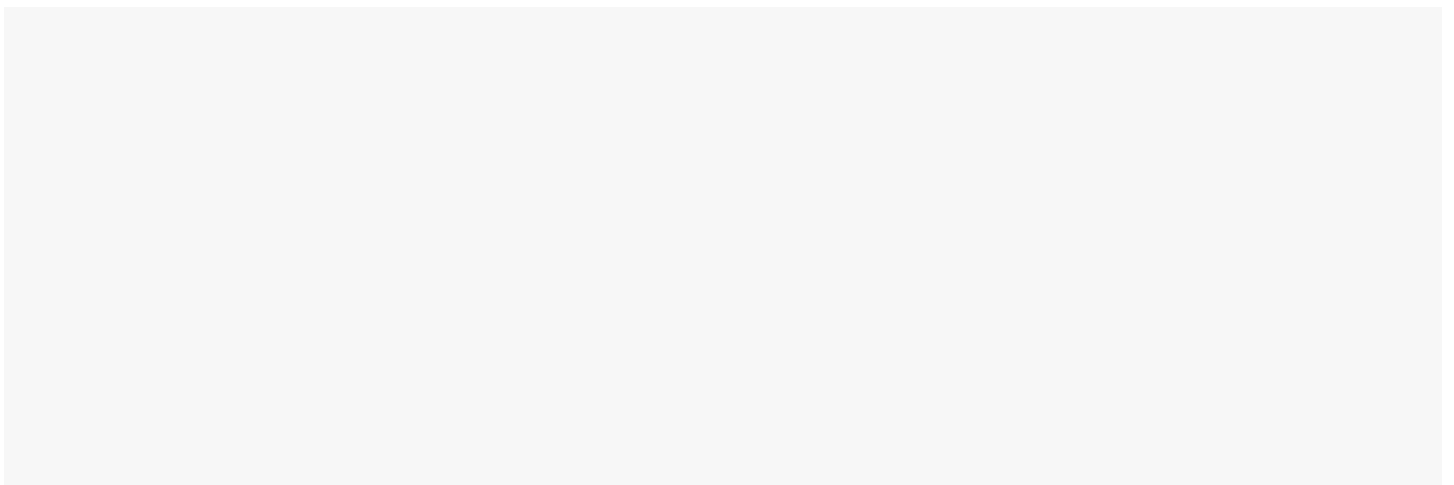
- Class definitions can be long, reducing readability in Jupyter notebooks. Using utilities like `register` functions as methods after class creation enables modular code blocks.

### 3.2.2. Models

- Introduces the `Module` class, a base class for all models. It has three methods:
  - `__init__` stores parameters.
  - `training_step` handles data and computes loss.
  - `configure_optimizers` specifies optimization methods.

## ✓ 3.2 Exercises

2. It is not possible to print `self.a` and `self.b` as without `save_hyperparameters`, both attributes will not be set during the initialization of the class.



```

import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l

class B(d2l.HyperParameters):
    def __init__(self, a, b, 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)

```



```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-1-232a760074f8> in <cell line: 12>()
    10     print('There is no self.c =', not hasattr(self, 'c'))
    11
--> 12 b = B(a=1, b=2, c=3)

<ipython-input-1-232a760074f8> in __init__(self, a, b, c)
     7 class B(d2l.HyperParameters):
     8     def __init__(self, a, b, c):
--> 9         print('self.a =', self.a, 'self.b =', self.b)
    10         print('There is no self.c =', not hasattr(self, 'c'))
    11

AttributeError: 'B' object has no attribute 'a'

```

Next steps: [Explain error](#)

## ✓ 3.4 Discussion

### 3.4.2 Defining the Loss Function:

- Introduces the squared loss function to measure how well the model's predictions match the true values. The implementation normalizes the true values and predicted values to have the same shape for proper calculation.

### 3.4.3 Defining the Optimization Algorithm:

- Introduces the Stochastic Gradient Descent (SGD) optimization method, which updates model parameters based on the gradient of the loss function.
- Defines a custom SGD class, which updates weights and biases after each minibatch and resets gradients before every new step.

## ✓ 3.4 Exercises

1.

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

```
class LinearRegressionScratch(d2l.Module): #
    """The linear regression model implemente
    def __init__(self, num_inputs, lr, sigma=
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.zeros((2, 1), requires
        self.b = torch.zeros(1, requires_grad
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
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_()
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```

@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,
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0: #
                self.clip_gradients(self.grad
            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.p
            self.val_batch_idx += 1

```

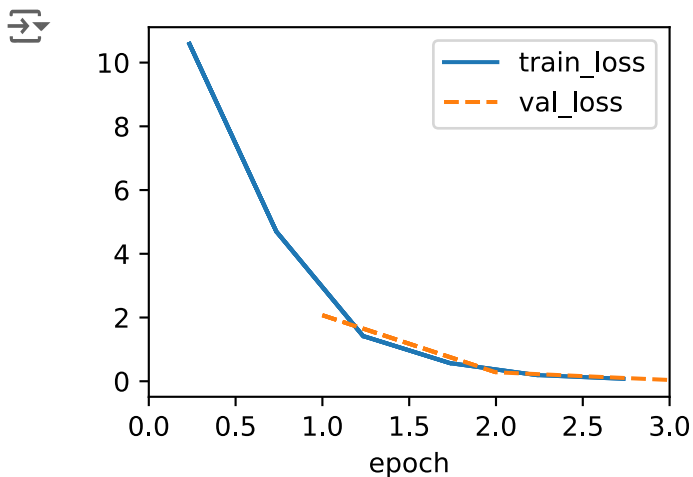
'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].

```

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)

```



```

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([ 0.1022, -0.1855])
error in estimating b: tensor([0.1957])

```

```
class LinearRegressionScratch(d2l.Module): #
    """The linear regression model implemente
    def __init__(self, num_inputs, lr, sigma=
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_
        self.b = torch.zeros(1, requires_grad
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
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_()
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
@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,
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].





```

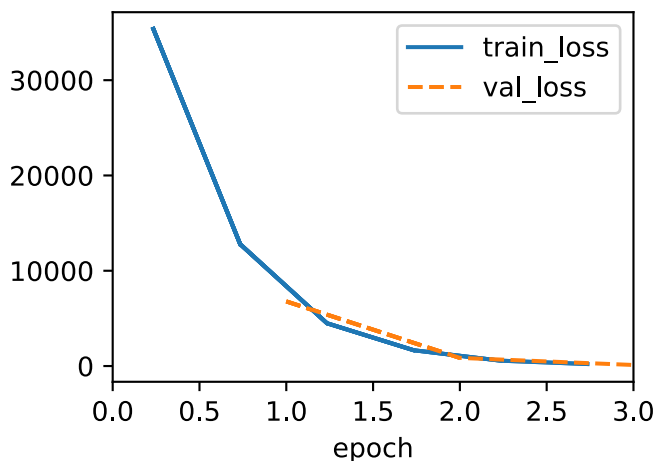
        if self.gradient_clip_val > 0: #
            self.clip_gradients(self.grad
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.p
            self.val_batch_idx += 1

```

```

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)

```



```

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([-6.7467, -13.0180])
error in estimating b: tensor([-1.4177])

```

4. Computing the second derivatives of the loss in linear regression can present several problems:

- The calculation of second derivatives can be computationally intensive, especially with large datasets, as it involves computing the Hessian matrix, which requires  $O(n^2)$  storage and  $O(n^3)$  time for inversion.
- The Hessian can be ill-conditioned, leading to numerical instability in optimization algorithms, which can cause convergence issues.
- Using second derivatives can make the model sensitive to noise, potentially leading to overfitting on training data.

To address these issues, some solutions are:

- Instead of calculating the exact Hessian, techniques like the Gauss-Newton approximation or L-BFGS can help estimate the second derivatives more efficiently and avoid direct computation of the full Hessian.
- Incorporating regularization techniques to stabilize the optimization process and reduce the risk of overfitting.
- For large datasets, mini-batch processing can be used to compute estimates of the Hessian using a subset of the data, thus reducing the computational burden.

5. The `reshape` method is needed in the loss function for the following reasons:

- When computing loss (e.g., in classification tasks), it's essential that the predicted outputs and the target labels have compatible shapes.
- For tasks such as image classification, output from a neural network may be a multi-dimensional tensor.
- Sometimes, different tensor shapes are needed for broadcasting rules to apply correctly during loss calculations. Reshaping can make sure that tensors align properly for operations like summation or averaging.
- When working with mini-batches, the shapes of predictions and targets must align.

## ✓ 4.1 Discussion

- Differentiates between "how much?" (regression) and "which category?" (classification) questions.
- Discusses distinction between predicting a single class and assigning probabilities to multiple classes.
- Explains cases where an example might belong to multiple categories.

### 4.1.1 Classification

- Classifies grayscale images into categories like "cat," "chicken," and "dog" with four pixel-based features.

#### 4.1.1.1 Linear Model

- Describes the need for one affine function per class output, requiring multiple weight and bias terms.
- Discusses how the classification can be viewed as a single-layer neural network where each output depends on all inputs.

#### 4.1.1.2 The Softmax

- Explains why simple vector-valued regression models don't always fit for classification due to nonnegativity and sum-to-one constraints.
- Introduces softmax, which converts outputs into probabilities by normalizing exponential transformations of inputs.

#### 4.1.2 Loss Function

- Introduces the need for a loss function that aligns with probabilistic outputs, using maximum likelihood estimation.

##### 4.1.2.1 Log-Likelihood

- Converts the problem of maximizing likelihood into minimizing the negative log-likelihood for more practical optimization.
- Introduces the cross-entropy loss, widely used for classification tasks, and explains why it is a bound on prediction error.

##### 4.1.2.2 Softmax and Cross-Entropy Loss

- Explains how the gradient of the softmax cross-entropy loss resembles the difference between predicted and actual class probabilities.

#### 4.1.3 Information Theory Basics

- Defines entropy as a measure of information and the minimum number of bits required to encode a message.
- Links predictability to compressibility, with less surprising data being easier to compress.

## ✓ 4.1 Exercises

1.1 The cross-entropy loss for softmax is:

$$l(y, \hat{y}) = - \sum_i y_i \log \hat{y}_i$$

The first derivative with respect to  $o_k$  is:

$$\frac{\partial l(y, \hat{y})}{\partial o_k} = \hat{y}_k - y_k$$

Differentiating again gives:

$$\frac{\partial^2 l(y, \hat{y})}{\partial o_k \partial o_j} = \begin{cases} \hat{y}_k(1 - \hat{y}_k) & \text{if } k = j \\ -\hat{y}_k \hat{y}_j & \text{if } k \neq j \end{cases}$$

1.2 For the softmax probabilities  $\hat{y}_i$ :

$$\text{Var}(\hat{y}_i) = \hat{y}_i(1 - \hat{y}_i)$$

The covariance between different classes is:

$$\text{Cov}(\hat{y}_i, \hat{y}_j) = -\hat{y}_i \hat{y}_j$$

The second derivative of the loss matches the variance-covariance structure of the softmax distribution:

- Variance:  $\hat{y}_i(1 - \hat{y}_i)$ .
- Covariance:  $-\hat{y}_i \hat{y}_j$ .

This connection shows that the curvature of the loss function reflects the uncertainty in the softmax outputs.

3. To transmit an integer in the range  $\{0, \dots, 7\}$  using ternary signals (PAM-3 with levels  $\{-1, 0, 1\}$ ), it is necessary to determine how many ternary units are required. Each ternary unit can represent three values  $\{-1, 0, 1\}$ . To find the number of ternary units  $n$  needed to represent 8 values (0 to 7), we solve the inequality:

$$3^n \geq 8$$

Since  $3^2 = 9$  is the smallest power that is greater than or equal to 8, 2 ternary units are needed.

Using ternary signals instead of binary can be beneficial for several reasons:

- Ternary encoding allows for more information to be transmitted with fewer units (2 units instead of 3 binary bits).
- Fewer transitions between signal levels can reduce power consumption and electromagnetic interference, improving signal integrity.
- Ternary systems can sometimes lead to simpler circuit designs since they may require fewer components for encoding and decoding.

## ✓ 4.2 Exercises

1.

```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

```
class FashionMNIST(d2l.DataModule): #@save
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transform
            transform
        self.train = torchvision.datasets.Fas
            root=self.root, train=True, trans
        self.val = torchvision.datasets.Fashi
            root=self.root, train=False, tran
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



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

```
⇒ (60000, 10000)
```

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

```
⇒ torch.Size([1, 32, 32])
```

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

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
@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,
        num_wo
```

'@save' is not an allowed annotation – allowed values include [@param, @title, @markdown].



```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

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

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

```
⇒ '121.06 sec'
```

It affects performance by increasing the computation time.

## ✓ 4.3 Discussion

### 4.3.1. The Classifier Class:

- Simplifies the process by averaging loss and accuracy over validation data batches.
- By default, the class uses stochastic gradient descent (SGD) as the optimizer, applied in minibatches, similar to linear regression.

### 4.3.2. Accuracy:

- Explains how classification accuracy is computed as the fraction of correct predictions.
- The classifier's accuracy is essential, though it is challenging to optimize directly due to its non-differentiability.
- Describes how accuracy is calculated by selecting the class with the highest predicted probability and comparing it with the true label using element-wise comparison.

## ✓ 4.3 Exercises

1.

- $L_v$ : The actual validation loss over the entire validation dataset.
- $L_v^q$ : The quick estimate of the validation loss computed by averaging losses across several minibatches.
- $l_v^b$ : The loss on the last minibatch of the validation data.
- $N_v$ : The total number of samples in the validation dataset.
- $N_b$ : The number of samples in a minibatch.

The quick estimate  $L_v^q$  is computed by averaging the minibatch losses over  $k$  minibatches (excluding the last minibatch). So, assuming  $k$  minibatches, each of size  $N_b$ , the estimate is:

$$L_v^q = \frac{1}{k} \sum_{i=1}^k l_v^i$$

The total validation loss  $L_v$  is computed as the weighted average of the losses from the minibatches used for the quick estimate  $L_v^q$  and the last minibatch  $l_v^b$ . If the last minibatch contains  $N_v - kN_b$  samples, the expression for  $L_v$  is:

$$L_v = \frac{kN_b}{N_v} L_v^q + \frac{N_v - kN_b}{N_v} l_v^b$$

## 2. Full Validation Loss:

$$L_v = \frac{1}{N_v} \sum_{i=1}^{N_v} l_v^i$$

Quick Estimate:

$$L_v^q = \frac{1}{k} \sum_{i=1}^k \frac{1}{N_b} \sum_{j \in B_i} l_v^j$$

Expected Value: Since minibatches are random samples:

$$E[L_v^q] = \frac{1}{k} \sum_{i=1}^k E \left[ \frac{1}{N_b} \sum_{j \in B_i} l_v^j \right] = L_v$$

Thus,  $E[L_v^q] = L_v$ , showing  $L_v^q$  is unbiased.

There are three possible reasons to use  $L_v$  instead of  $L_v^q$ :

- $L_v^q$  has higher variance since it's based on fewer samples.
- $L_v$  uses the entire dataset, providing a more reliable estimate.
- Full dataset evaluations reduce the impact of outliers.

## ✓ 4.4 Discussion

### 4.4.1. The Softmax:

- The softmax function maps input scalars to probabilities by normalizing the values.

### 4.4.2. The Model:

- Softmax regression model represents inputs as fixed-length vectors. For image data, images are flattened into vectors (length 784).
- The model parameters (weights and biases) are initialized using Gaussian noise and zeros, respectively.

### 4.4.3. The Cross-Entropy Loss:

- Cross-entropy is one of the most common loss functions for classification problems, taking the negative log-likelihood of the predicted probability assigned to the true label.
- Efficiently selects the predicted probability corresponding to the correct class using indexing.
- Implements the cross-entropy loss by averaging the logarithms of selected probabilities.

## ✓ 4.4 Exercises

3. In softmax regression, returning the most likely label isn't always the best idea, especially in critical areas like medical diagnosis. In medical diagnosis, predicting the most likely disease without considering uncertainty can be dangerous, especially if the second-most likely label is life-threatening. Furthermore, softmax probabilities might be close, indicating uncertainty. Always choosing the highest probability ignores this uncertainty, leading to overconfidence in predictions.

There are various possible approaches to address these issues. For example, instead of always returning the most likely label, using a confidence threshold. If no label meets the threshold, further tests or a human review could be recommended. It is also possible to present the top-k probable labels, giving a range of possibilities for further investigation.

4. With a large vocabulary in softmax regression for word prediction, the following problems may arise:

- Calculating softmax over a large number of words can be computationally expensive and slow.
- Storing the parameters for each word in the vocabulary requires a significant amount of memory.
- Words that appear infrequently may result in poor predictions due to insufficient training data.
- A large vocabulary increases the risk of overfitting, especially if some words have limited context in the training data.

## ✓ 5.1 Discussion

### 5.1.1. Hidden Layers

- Points out that simple pixel brightness analysis is insufficient for tasks like distinguishing between cats and dogs, necessitating more sophisticated models.
- Mentions past methods for handling nonlinearity, such as decision trees and kernel methods, indicating a long-standing interest in this area.

### 5.1.1.2. Incorporating Hidden Layers

- Describes how adding hidden layers allows for more complex representations and outputs in a multilayer perceptron.

### 5.1.1.3. From Linear to Nonlinear

- Notes that hidden layers introduce additional parameters, but without nonlinear activation functions, they do not enhance model expressiveness.



- Introduces the necessity of nonlinear activation functions to enable the model to capture complex patterns and prevent it from collapsing to a linear model.

#### 5.1.1.4. Universal Approximators

- Discusses the capability of deep networks to approximate any function given sufficient complexity.

#### 5.1.2.1. ReLU Function

- Describes ReLU's simplicity and effectiveness, making it the most commonly used activation function in practice.

#### 5.1.2.2. Sigmoid Function

- Describes challenges in optimization due to the vanishing gradients associated with sigmoid activation, leading to its reduced usage in hidden layers.

#### 5.1.2.3. Tanh Function

- Highlights the range and symmetry of the tanh function, noting its superior output range compared to sigmoid.
- Describes how tanh behaves similarly to sigmoid regarding its derivatives, impacting learning dynamics.

## ✓ 5.1 Exercises

1. In a linear deep network, each layer performs a linear transformation on its input. Without a nonlinearity (like  $\sigma$ ), stacking multiple linear layers is equivalent to applying a single linear transformation. This is because the composition of linear transformations is still a linear transformation. Hence, adding more layers does not increase the network's expressive power.

Consider a simple linear deep network with two layers:

1. First layer:  $\mathbf{h}_1 = \mathbf{W}_1 \mathbf{x}$
2. Second layer:  $\mathbf{h}_2 = \mathbf{W}_2 \mathbf{h}_1 = \mathbf{W}_2 (\mathbf{W}_1 \mathbf{x})$

The output of the network is:  $\mathbf{y} = \mathbf{W}_2 \mathbf{W}_1 \mathbf{x}$

Since the product of two matrices  $\mathbf{W}_2 \mathbf{W}_1$  is still a single matrix, this is equivalent to a one-layer network with weight matrix  $\mathbf{W} = \mathbf{W}_2 \mathbf{W}_1$ . Thus, no matter how many layers are added, the entire network behaves like a single linear layer.

2. The Parametric ReLU (pReLU) activation function is defined as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where  $\alpha$  is a learned parameter.

The derivative of the pReLU function with respect to  $x$  is computed piecewise:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases}$$

3. The Swish activation function is defined as:

$$f(x) = x \cdot \text{sigmoid}(\beta x)$$

To compute the derivative of  $f(x) = x \cdot \text{sigmoid}(\beta x)$  the product rule can be used to obtain:

$$f'(x) = \text{sigmoid}(\beta x) + x \cdot [\text{sigmoid}(\beta x)(1 - \text{sigmoid}(\beta x)) \cdot \beta]$$

Simplifying:

$$f'(x) = \text{sigmoid}(\beta x) + \beta x \cdot \text{sigmoid}(\beta x)(1 - \text{sigmoid}(\beta x))$$

This expression combines both the sigmoid function and its derivative to describe how the Swish function changes with respect to  $x$ .

## ✓ 5.2 Discussion

- MLPs are not significantly more complex than linear models. The main difference lies in stacking multiple layers for enhanced capability.

## ✓ 5.2 Exercises

1.

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

```
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))

```

```

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

```

```

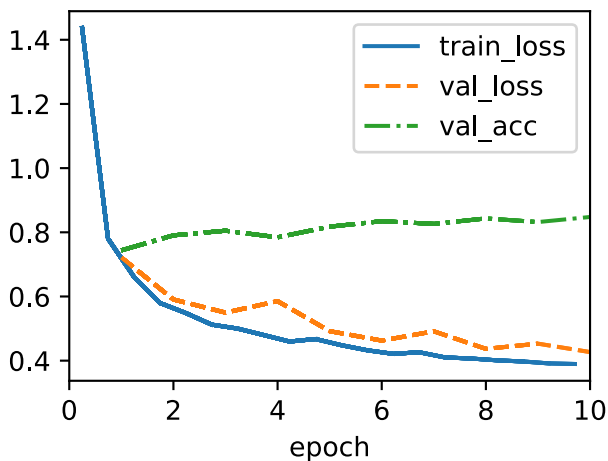
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2

```

```

model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=100, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)

```



2.

```

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))
        self.W3 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b3 = nn.Parameter(torch.zeros(num_outputs))

```

```

def relu(X):
    a = torch.zeros_like(X)

```

```
return torch.max(X, a)
```

```
@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H1 = relu(torch.matmul(X, self.W1) + self.b1)
    H2 = relu(torch.matmul(H1, self.W2) + self.b2)
    return torch.matmul(H2, self.W3) + self.b3
```

```
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)
```



```
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-82-30c732ea413e> in <cell line: 4>()
      2 data = d2l.FashionMNIST(batch_size=256)
      3 trainer = d2l.Trainer(max_epochs=10)
----> 4 trainer.fit(model, data)

-----
5 frames
<ipython-input-81-f54ed5eb35a6> in forward(self, X)
      4     H1 = relu(torch.matmul(X, self.W1) + self.b1)
      5     H2 = relu(torch.matmul(H1, self.W2) + self.b2)
----> 6     return torch.matmul(H2, self.W3) + self.b3

RuntimeError: mat1 and mat2 shapes cannot be multiplied (256x10 and 256x10)
```

Next steps: [Explain error](#)

3. Inserting a hidden layer with a single neuron limits the network's ability to model complex relationships, creates bottlenecks, increases the risk of overfitting, and may hinder effective learning through inadequate gradients. It's generally better to use multiple neurons in hidden layers to enhance expressiveness and learning capacity.

## ✓ 5.3 Discussion

- This section discusses the importance of both forward and backward propagation in training neural networks, emphasizing that while forward propagation is straightforward, understanding backpropagation is crucial for deeper insights into deep learning.

### 5.3.1. Forward Propagation

- Forward propagation involves calculating and storing intermediate variables as data flows through a neural network, from the input to the output layer.

## ✓ 5.3 Discussion

- This section discusses the importance of both forward and backward propagation in training neural networks, emphasizing that while forward propagation is straightforward, understanding backpropagation is crucial for deeper insights into deep learning.

### 5.3.1. Forward Propagation

- Forward propagation involves calculating and storing intermediate variables as data flows through a neural network, from the input to the output layer.

### 5.3.3. Backpropagation

- Backpropagation is introduced as the method for calculating gradients of network parameters by traversing the network in reverse order using the chain rule.
- The steps to compute gradients of the objective function, the loss term, and the regularization term are outlined, illustrating how to derive gradients for both output and hidden layer parameters.

### 5.3.4. Training Neural Networks

- The need to retain intermediate values from forward propagation until backpropagation is complete is discussed, which increases memory requirements during training.
- Larger batch sizes and deeper networks can lead to out-of-memory errors.

## 5.3 Exercises

1. If the inputs  $X$  to a scalar function  $f$  are  $n \times m$  matrices, the gradient of  $f$  with respect to  $X$  will have the same dimensionality as  $X$ .
  - The function  $f$  takes an  $n \times m$  matrix as input and returns a scalar value.
  - The gradient of  $f$  with respect to  $X$  represents how  $f$  changes with respect to small changes in each element of  $X$ . Since  $X$  has  $n \times m$  elements, the gradient will also have  $n \times m$  elements.

Thus, the dimensionality of the gradient of  $f$  with respect to  $X$  is  $n \times m$ .

4. When computing second derivatives in backpropagation:
  - The computational graph includes both the original operations and the gradients from the first backpropagation, effectively doubling the complexity.
  - Each operation that contributed to the gradient will have new nodes representing the second derivatives.
  - Relationships among nodes become more intricate, as the second derivative measures how the gradient changes with respect to inputs.
  - Calculating second derivatives is more expensive than first derivatives due to the need for an additional backward pass.
  - If the original backpropagation takes  $O(T)$ , computing second derivatives might take around  $O(k \cdot T)$ , where  $k$  is a factor based on function complexity.

5.1 Yes, it is possible to partition the computational graph across multiple GPUs using data parallelism and model parallelism. Data parallelism consists in splitting training data into smaller batches across GPUs, each computing gradients on its subset. Model parallelism consists of distributing different parts of the model across GPUs.

5.2 Advantages:

- Allows training larger models or using bigger batch sizes.
- Parallel computations reduce overall training time.
- Fully leverages multiple GPUs.

Disadvantages:

- Adds challenges in data handling and synchronization.
- Data transfer between GPUs can slow down training.
- Larger batches can affect convergence and generalization.