2. Preliminaries

2.1. Data Manipulation

2.1.1. Getting Started

In [11]:
```
import torch
```

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

Out[12]:
```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

In [13]:
```
x.numel()  ## number of elements
```

Out[13]:
```
12
```

In [14]:
```
x.shape  ## size of tensor
```

Out[14]:
```
torch.Size([12])
```

In [15]:
```
X = x.reshape(3, 4)
X
```

Out[15]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

In [16]:
```
torch.zeros((2, 3, 4))
```

Out[16]:
```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

In [17]:
```
torch.ones((2, 3, 4))
```

Out[17]:
```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

In [18]:
```
torch.randn(3, 4)
```

Out[18]:
```
tensor([[ 0.2673, -0.0161, -2.0190, -0.8120],
        [ 1.3601,  0.9666,  0.1938,  1.5541],
        [ 0.1794,  0.2149, -1.6761,  0.2586]])
```

In [19]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

Out[19]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

2.1.2. Indexing and Slicing

In [22]:
```
X[-1], X[1:3]
```

Out[22]:
```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

In [23]:
```
X[1, 2] = 17
X
```

Out[23]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

In [24]:
```
X[:2, :] = 12
X
```

Out[24]:
```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

2.1.3. Operations

In [25]:
```
torch.exp(x)
```

Out[25]:
```
tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
        162754.7969, 162754.7969, 162754.7969,   2980.9580,   8103.0840,
         22026.4648,  59874.1406])
```

In [26]:
```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

Out[26]:
```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

In [48]:
```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

Out[48]:
```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

In [49]:
```
X == Y
```

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

In [50]:
```
X > Y
```

Out[50]:
```
tensor([[False, False, False, False],
        [ True,  True,  True,  True],
        [ True,  True,  True,  True]])
```

In [51]:
```
X < Y
```

Out[51]:
```
tensor([[ True, False,  True, False],
        [False, False, False, False],
        [False, False, False, False]])
```

In [29]:
```
X.sum()
```

Out[29]:
```
tensor(66.)
```

2.1.4. Broadcasting

In [30]:
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

Out[30]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

In [32]:
```
a + b
## 3*1 + 1*2 -> broadcasting produces a larger 3*2 matrix by replicating
## matrix a along columns (3,2) and matrix b along rows (3,2)
```

Out[32]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

2.1.5. Saving Memory

In [36]:
```
before = id(Y)
Y = Y + X
id(Y) == before
## id() function: exact address of referenced object in memory
## Y = Y + X, so id(Y) points to a different location
```

Out[36]:
```
False
```

In [44]:
```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
## allocating Z (initialize) and overwrite the values of Z
```

```
id(Z): 130564857457232
id(Z): 130564857457232
```

In [40]:
```
before = id(X)
X += Y
id(X) == before
```

Out[40]:
```
True
```

2.1.6. Conversion to Other Python Objects

In [41]:
```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

Out[41]:
```
(numpy.ndarray, torch.Tensor)
```

In [42]:
```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[42]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```

2.2. Data Preprocessing

2.2.1. Reading the Dataset

In [1]:
```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Price
            NA,NA,127500
            2,NA,106000
            4,Slate,178100
            NA,NA,140000''')
```

In [2]:
```
import pandas as pd

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

```
         NumRooms RoofType   Price
0              NA      NaN  127500
1               2      NaN  106000
2               4    Slate  178100
3              NA      NaN  140000
```

2.2.2. Data Preparation

In [3]:
```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms_           2  NumRooms_           4  NumRooms_           NA  \
0                  False                  False                    True
1                   True                  False                   False
2                  False                   True                   False
3                  False                  False                    True

   NumRooms_nan  RoofType_Slate  RoofType_nan
0         False           False          True
1         False           False          True
2         False            True         False
3         False           False          True
```

In [4]:
```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms_           2  NumRooms_           4  NumRooms_           NA  \
0                  False                  False                    True
1                   True                  False                   False
2                  False                   True                   False
3                  False                  False                    True

   NumRooms_nan  RoofType_Slate  RoofType_nan
0         False           False          True
1         False           False          True
2         False            True         False
3         False           False          True
```

2.2.3. Conversion to the Tensor Format

In [10]:
```
import torch

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

Out[10]:
```
(tensor([[0., 0., 1., 0., 0., 1.],
         [1., 0., 0., 0., 0., 1.],
         [0., 1., 0., 0., 1., 0.],
         [0., 0., 1., 0., 0., 1.]], dtype=torch.float64),
 tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

2.3. Linear Algebra

2.3.1. Scalars

In [16]:
```
import torch
```

In [17]:
```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

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

Out[17]:
```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

2.3.2. Vectors

x = torch.arange(3) x

In [22]:
```
x[2]
```

Out[22]:
```
tensor(2)
```

In [23]:
```
x.shape
```

Out[23]:
```
torch.Size([3])
```

2.3.3. Matrices

In [25]:
```
A = torch.arange(6).reshape(3, 2)
A
```

Out[25]:
```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

In [33]:
```
A.T  ## Transpose
```

Out[33]:
```
tensor([[0., 3.],
        [1., 4.],
        [2., 5.]])
```

In [34]:
```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A == A.T
```

Out[34]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

2.3.4. Tensors

In [35]:
```
torch.arange(24).reshape(2, 3, 4)
```

Out[35]:
```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

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

2.3.5. Basic Properties of Tensor Arithmetic

In [36]:
```
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
```

Out[36]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  2.,  4.],
         [ 6.,  8., 10.]]))
```

In [39]:
```
A * B  ## elementwise product of two matrices are Hadamard product
```

Out[39]:
```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

In [40]:
```
a = 2  ## tensor * scalar -> same shape as original
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Out[40]:
```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

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

2.3.6. Reduction

In [41]:
```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

Out[41]:
```
(tensor([0., 1., 2.]), tensor(3.))
```

In [46]:
```
A
```

Out[46]:
```
tensor([[0., 1., 2.],
        [3., 4., 5.]])
```

In [47]:
```
A.shape, A.sum()
```

Out[47]:
```
(torch.Size([2, 3]), tensor(15.))
```

In [53]:
```
A.shape, A.sum(axis=0).shape ## (3, 5, 7)
```

Out[53]:
```
(torch.Size([2, 3]), torch.Size([3]))
```

In [54]:
```
A.shape, A.sum(axis=1).shape ## (3, 12)
```

Out[54]:
```
(torch.Size([2, 3]), torch.Size([2]))
```

In [52]:
```
A.sum(axis=[0, 1]) == A.sum()
```

Out[52]: tensor(True)

In [55]:
```
A.mean(), A.sum() / A.numel()
```

Out[55]: (tensor(2.5000), tensor(2.5000))

In [56]:
```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

Out[56]: (tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))

2.3.7. Non-Reduction Sum

In [57]:
```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

Out[57]:
```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

In [58]:
```
A / sum_A
```

Out[58]:
```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

In [59]:
```
A.cumsum(axis=0) ## cumulative sum of A along some axis
```

Out[59]:
```
tensor([[0., 1., 2.],
        [3., 5., 7.]])
```

In [60]:
```
A.cumsum(axis=1)
```

Out[60]:
```
tensor([[ 0.,  1.,  3.],
        [ 3.,  7., 12.]])
```

2.3.8. Dot Products

In [61]:
```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

Out[61]: (tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))

In [63]:
```
x * y
```

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

In [66]:
```
torch.sum(x * y)  ## dot product xTy = <x,y> = sigma xy
```

Out[66]: tensor(3.)

2.3.9. Matrix–Vector Products

In [67]:
```
A.shape, x.shape, torch.mv(A, x), A@x
```

Out[67]: (torch.Size([2, 3]), torch.Size([3]), tensor([5., 14.]), tensor([5., 14.]))

2.3.10. Matrix–Matrix Multiplication

In [69]:
```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

Out[69]:
```
(tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]),
 tensor([[ 3.,  3.,  3.,  3.],
         [12., 12., 12., 12.]]))
```

In [70]:
```
torch.matmul(A, B)
```

Out[70]:
```
tensor([[ 3.,  3.,  3.,  3.],
        [12., 12., 12., 12.]])
```

2.3.11. Norms

In [82]:
```
## norm of a vector tells us how big it is.
## A norm is a function
## that maps a vector to a scalar and satisfies the following three properties:
## 1. Given any vector x, if we scale the vector by a scalar a, norm scales
##    \\ax\\ = \a\\\x\\
## 2. For any vectors x and y: norms satisfy triangle inequality:
##    \\x+y\\ <= \\x\\+\\y\\
## 3. The norm of vector is nonnegative and it only vanishes if vector is zero:
##    \\x\\ > 0 for all x!=0
```

In [76]:
```
## l2 norm = Euclidean length of vector
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

Out[76]:
```
tensor(5.)
```

In [78]:
```
## l1 norm = absolute value
torch.abs(u).sum()
```

Out[78]:
```
tensor(7.)
```

In [81]:
```
torch.norm(torch.ones((4, 9)))
```

Out[81]:
```
tensor(6.)
```

2.5. Automatic Differentiation

2.5.1. A Simple Function

In [96]:
```
x = torch.arange(4.0)
x
```

Out[96]:
```
tensor([0., 1., 2., 3.])
```

In [97]:
```
# Can also create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad  # The gradient is None by default
```

In [98]:
```
y = 2 * torch.dot(x, x)
y
```

Out[98]:
```
tensor(28., grad_fn=<MulBackward0>)
```

In [99]:
```
y.backward()
x.grad  ## gradient of y=2xTx -> dy/dx = 4x
```

Out[99]:
```
tensor([ 0.,  4.,  8., 12.])
```

In [101…
```
x.grad == 4 * x
```
Out[101… tensor([True, True, True, True])

In [103…
```
x.grad.zero_()  # Reset the gradient
y = x.sum()
y.backward()
x.grad
```
Out[103… tensor([1., 1., 1., 1.])

2.5.2. Backward for Non-Scalar Variables

In [106…
```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))  # Faster: y.sum().backward()
x.grad
```
Out[106… tensor([0., 2., 4., 6.])

2.5.3. Detaching Computation

In [111…
```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```
Out[111… tensor([True, True, True, True])

In [112…
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```
Out[112… tensor([True, True, True, True])

2.5.4. Gradients and Python Control Flow

In [114…
```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

In [116…
```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

In [118…
```
a.grad == d / a ## linear function -> f(a)/a == gradient
```
Out[118… tensor(True)

3. Linear Neural Networks for Regression

3.1. Linear Regression

3.1.2. Vectorization for Speed

In [163…
```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

In [157…
```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

In [158…
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

Out[158… '0.05077 sec'

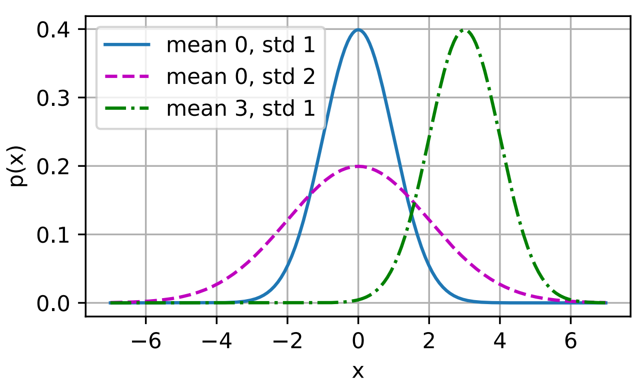
In [159…
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

Out[159… '0.00012 sec'

3.1.3. The Normal Distribution and Squared Loss

In [160…
```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

In [165…
```
# 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

3.2.1. Utilties

In [166…

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

In [167…

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

a = A()
```

In [170…

```
# decorate method by add_to_Class with class A as its argument
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

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

In [173…

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

In [174…

```
# 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)
```

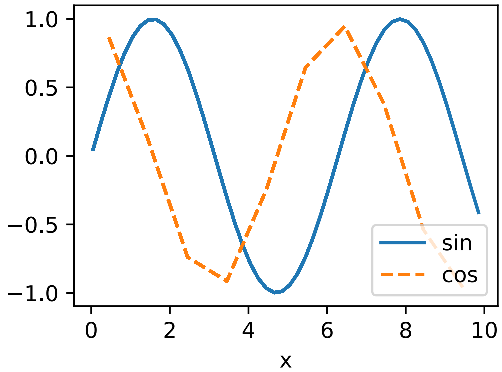
```
self.a = 1 self.b = 2
There is no self.c = True
```

In [175…

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

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

In [176…

```
board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)
```



3.2.2. Models

In [178…

```
import torch.nn as nn
```

In [179…

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

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

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

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

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

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

    def configure_optimizers(self):
        raise NotImplementedError
```

In [180…

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

```
    def get_dataloader(self, train):
        raise NotImplementedError

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

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

In [183…

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

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

    def prepare_model(self, model):
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        self.model = model

    def fit(self, model, data):
        self.prepare_data(data)
        self.prepare_model(model)
        self.optim = model.configure_optimizers()
        self.epoch = 0
        self.train_batch_idx = 0
        self.val_batch_idx = 0
        for self.epoch in range(self.max_epochs):
            self.fit_epoch()

    def fit_epoch(self):
        raise NotImplementedError
```

3.4. Linear Regression Implementation from Scratch

3.4.1. Defining the Model

In [184…

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

In [185…

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

3.4.2. Defining the Loss Function

In [188…

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

3.4.3. Defining the Optimization Algorithm

In [190…

```
# linear regression has closed-form solution
# minibatch SGD
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_()
```

In [191…

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

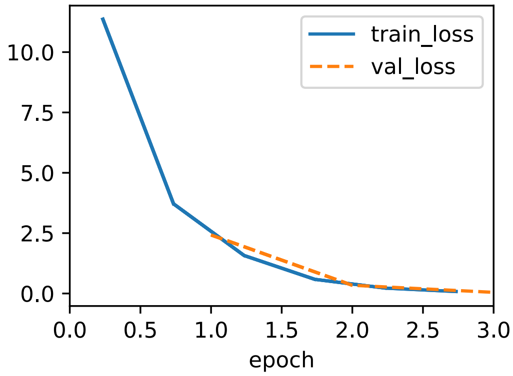
3.4.4. Training

In [192…

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

@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:  # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

In [193…

```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



In [194…

```
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([ 0.0753, -0.1851])
error in estimating b: tensor([0.2281])
```

4. Linear Neural Networks for Classification

4.1. Softmax Regression

4.2. The Image Classification Dataset

In [195…

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

d2l.use_svg_display()
```

4.2.1. Loading the Dataset

In [197…

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

In [198…

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

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to
../data/FashionMNIST/raw/train-images-idx3-ubyte.gz
100%|██████████████████████████| 26421880/26421880 [00:10<00:00, 2481979.44it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to
../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz
100%|█████████████████████████████████| 29515/29515 [00:00<00:00, 108773.00it/s]
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████████████████████████| 4422102/4422102 [00:09<00:00, 487071.20it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████████████████████████████| 5148/5148 [00:00<00:00, 28410890.78it/s]
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
```

Out[198…

```
(60000, 10000)
```

In [199…

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

Out[199…

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

In [200…

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

4.2.2. Reading a Minibatch

In [201…

```
@d2l.add_to_class(FashionMNIST)  #@save
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

In [202…

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

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

In [203…

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

Out[203…

```
'1.44 sec'
```

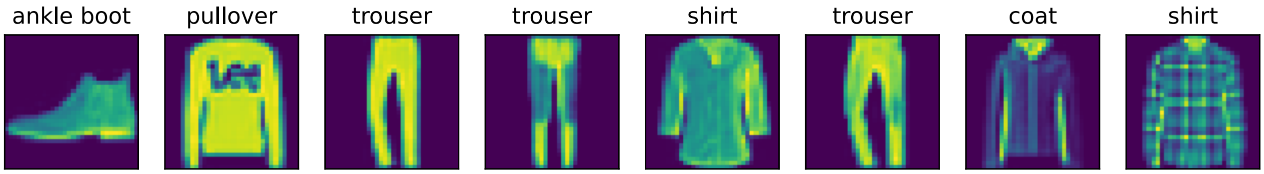
4.2.3. Visualization

In [204…

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

In [205…

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



4.3. The Base Classification Model

4.3.1. The Classifier Class

In [211…

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

In [212…

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

4.3.2. Accuracy

In [213…

```
# The result is tensor containing entries of 0(F) and 1(T). Sum is the number of correct predictions
```

In [214…

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

4.4. Softmax Regression Implementation from Scratch

4.4.1. The Softmax

In [215…

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

Out[215…

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

In [216…

```
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
```

In [217…

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

Out[217…

```
(tensor([[0.2189, 0.2594, 0.1736, 0.1528, 0.1954],
         [0.2245, 0.2166, 0.1560, 0.1575, 0.2454]]),
 tensor([1.0000, 1.0000]))
```

4.4.2. The Model

In [218…
```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs),
                              requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

In [219…
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

4.4.3. The Cross-Entropy Loss

In [230…
```
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]  # 0,0 , 1,2
```

Out[230…
```
tensor([0.1000, 0.5000])
```

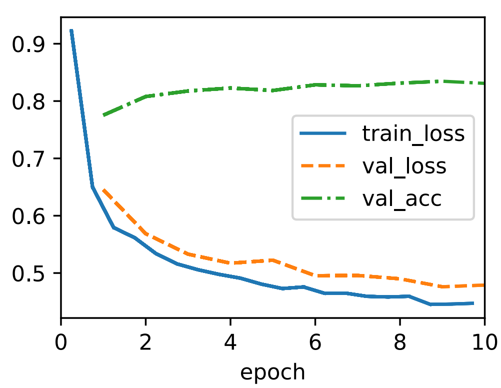
In [221…
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

Out[221…
```
tensor(1.4979)
```

In [222…
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

4.4.4. Training

In [223…
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



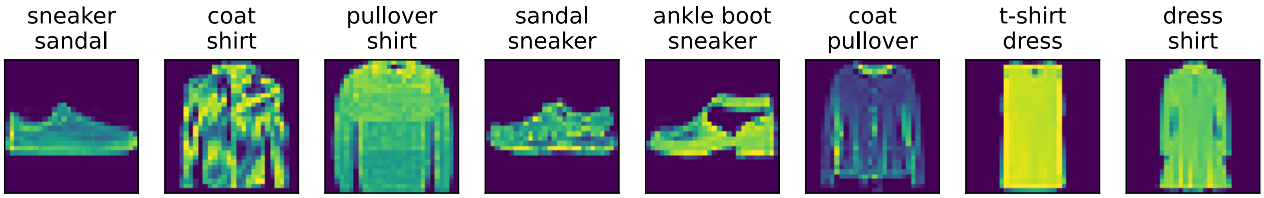
4.4.5. Prediction

In [224…
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

Out[224… torch.Size([256])

In [225…

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



5. Multilayer Perceptrons

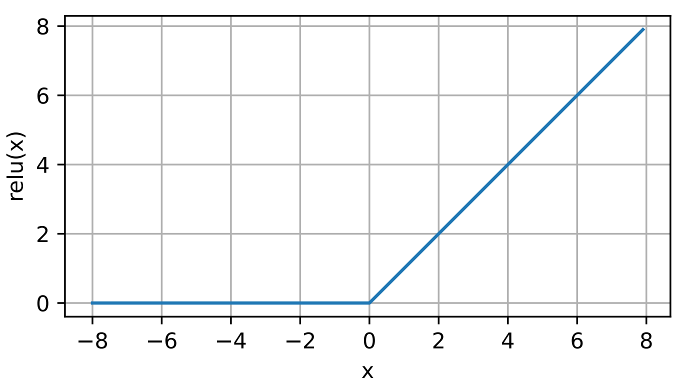
5.1. Multilayer Perceptrons

5.1.1. Hidden Layers

5.1.2. Activation Functions

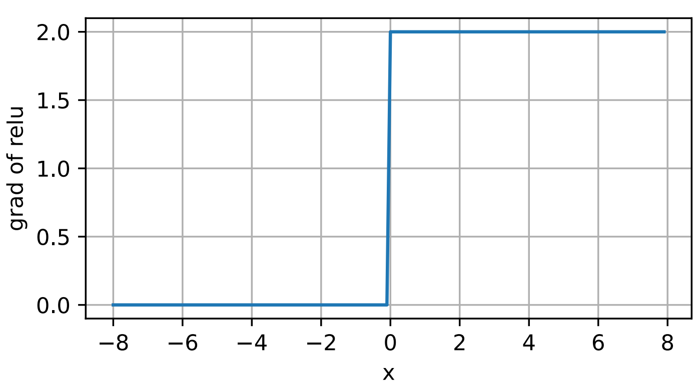
In [239…

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



In [241…

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

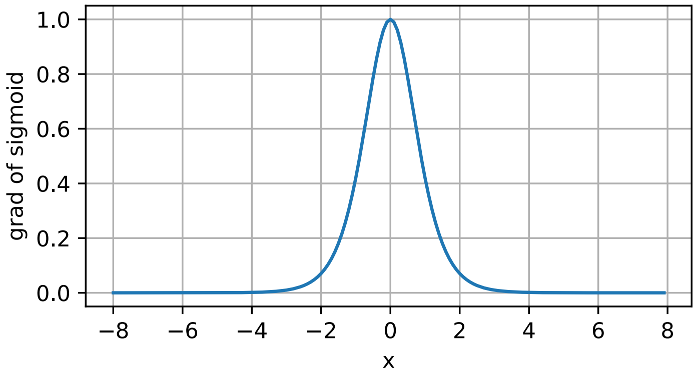


In [242…

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

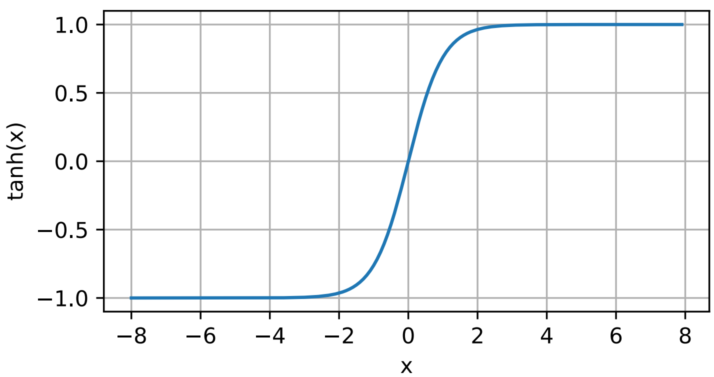
In [246…

```
## derivative of Sigmoid
# 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))
```



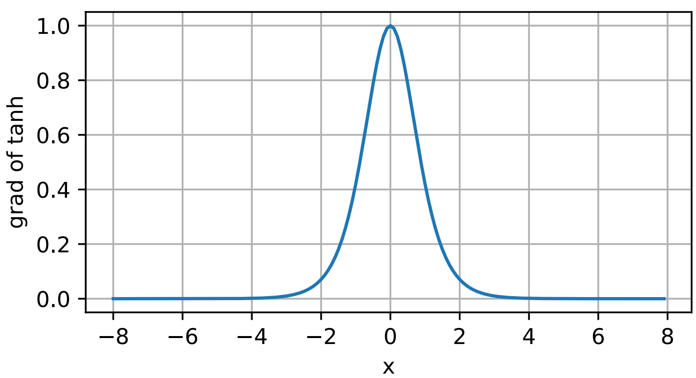
In [245…

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



In [248…

```
## derivative of Tanh
# 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

5.2.1. Implementation from Scratch

In [249…

```
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))
```

In [250…

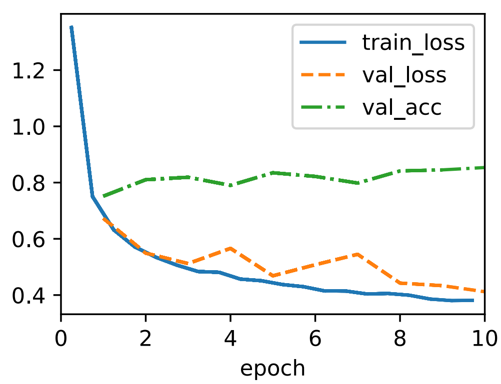
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

In [251…

```
@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
```

In [252…

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



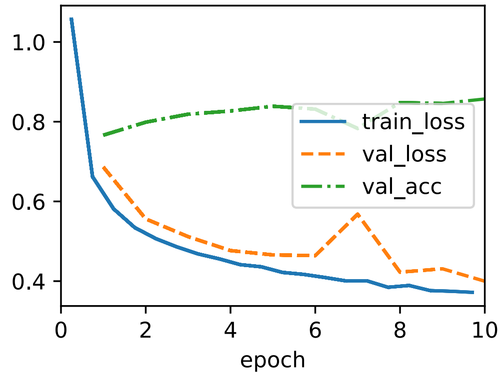
5.2.2. Concise Implementation

In [253…

```
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))
```

In [254…

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



5.3. Forward Propagation, Backward Propagation, and Computational Graphs

#. Discussion

2.

2.1.

In python, there is no operation such as malloc and free. Therefore, to allocate the memory first, it is important to initialize tensor (in member variable), and then overwrite.

2.2.

Rather than alternating NaN data into mean(), it may be better to calculate Euclidean Distance to each dataset and alternate with the closest value, or just drop by dropna.

2.3.

In tensor, axis is important; axis=0 is row and axis=1 is column in 2-dimensional matrix.

2.4.

When we create some auxiliary intermediate term that we don't want to compute gradient, we need to detach the respective computational graph from the final result

3.

3.1.

Mean squared error can be expressed by normal distribution. Maximum likelihood of normal distribution turns into minimization by negative log-likelihood.

3.2.

@add_to_class: the method is able to access the member variables of A just as we would expect had it been included as part of A's definition. @decotrator_: allow to wrap another function as an input and modify its behavior without altering the wrapped function's code

3.4.

Training step: initialize parameters(w,b) -> repeat until done (compute gradient / update parameters)

4.

4.1.

Cross Entropy는 이론적으로 두 확률 분포 사이의 차이를 측정하는 개념. Cross Entropy Loss는 이 개념을 실제 모델 학습에 적용하여, 모델의 예측이 실제 레이블과 얼마나 잘 일치하는지를 평가하는 손실 함수. (y와 y^의 차이를 최소화)

4.2.

[1, 32, 32] shape -> c * h * w (c=1) a data iterator reads a minibatch of data with size batch_size (here, 64). We also randomly shuffle the examples for the training data iterator.

4.3.

Y_hat = self(*batch[:-1]) batch의 구조: X, y = batch, Y_hat의 second dimension은 prediction scores -> largest index (preds)를 반환

4.4.

It will be better to normalize the data input first to prevent numerical instabilities. (Also to make loss positive)

5.

5.1.

tanh(x)+1=(1-exp(-2x))/(1+exp(-2x))+1=2/(1+exp(-2x))=2sigmoid(2x) -> just adding affine layer -> identical

5.2.

torch.randn(num_inputs, num_hiddens)는 평균이 0이고 표준 편차가 1인 정규분포에서 무작위로 값을 샘플링함. 이 값이 너무 클 수 있기 때문에, sigma라는 작은 값을 곱해 가중치 초기화를 더 적절한 범위로 조정하는 것.

5.3.

Forward propagation and backpropagation are interdependent, and training requires significantly more memory than prediction. Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer. The objective of backpropagation is to calculate the gradients.