```
import math
import time
import torch
from torch import nn
import numpy as np
import os
import pandas as pd
from d2l import torch as d2l
from matplotlib import pyplot as plt
import torchvision
from torchvision import transforms
#==============================2.1.1==================================
==

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

x.numel()

X = x.reshape(3,4)

Y = torch.zeros((2,3,4))

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

torch.randn(3,4)

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

#print(Z)

#==============================2.1.2==================================
==

#print(X[-1], X[1:3])

X[1,2] = 17

X[:2, :] = 12

#print(X)

#==============================2.1.3==================================
=

x = torch.exp(x)

x = torch.tensor([1.0, 2, 4, 8])
```

```
y = torch.tensor([2, 2, 2, 2])

#print(x + y, x - y, x * y, x / y, x ** y)

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]])
Xn, Yn = torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)

#print(Xn, Yn)

#print(X == Y)

#print(X.sum())

#==============================2.1.4==================================
==

a = torch.arange(5).reshape((5,1))
b = torch.arange(3).reshape((1,3))

#print(a,b)

#print(a + b)

#============================2.1.5====================================
===

before = id(Y)
Y = Y + X
#print(id(Y) == before)

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

before = id(X)
X += Y
#print(id(X) == before)

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

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

```
# tensors create a storage method for data storage than can be operated on and
# manipulated in a way that is helpful for deep learning.

# ============================== 2.2
==================================

# ============================== 2.2.1 ==================================

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''')

data = pd.read_csv(data_file)
#print(data)
data['NumRooms'] = pd.to_numeric(data['NumRooms'], errors='coerce')

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

inputs = inputs.fillna(inputs.mean())
#print(inputs)

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

# using pandas alongside tensors will allow for greater control
# over the data and allow us to work around missing values
# without too much effort

# ============================ 2.3 ==============================

# ============================ 2.3.1 ==============================

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

#print(x + y, x * y, x / y, x**y)

# ============================ 2.3.2 ==============================
```

```
x = torch.arange(3)

#print(x, x[2], len(x), x.shape)

#============================ 2.3.3 ==============================

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

#print(A, A.T)

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

#============================ 2.3.4 ==============================

#print(torch.arange(24).reshape(2, 3, 4))

A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()
#print(A, '\n', A+B, '\n', A*B)

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

#============================ 2.3.6 ==============================

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

#print(A.shape, A.sum(), A.sum(axis=0).shape, A.sum(axis=1).shape)

#print(A.sum(axis=[0,1]) == A.sum())

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

# ============================ 2.3.7 ==============================

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

#print(A / sum_A, A.cumsum(axis=0))

# ============================ 2.3.8 ==============================

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

```
#print(x,'\n', y, '\n' ,torch.dot(x,y))

#print(torch.sum(x * y))

# ============================== 2.3.9 ==================================

#print(A.shape, x.shape, torch.mv(A, x), A@x)

# ============================== 2.3.10 ==============================

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

# ============================== 2.3.11 ==============================

u = torch.tensor([3.0, -4.0])
#print(torch.norm(u), torch.abs(u).sum(), torch.norm(torch.ones((4, 9))))

# the linear algebra operations available in pytorch are very useful
# to deep learning as when using weights and biases these are represented as
# matrices which we can easily manipulate and perform calculations with to
# perform forward and back propagation.

# ============================== 2.5
==================================

# ============================== 2.5.1 ==================================

x = torch.arange(4.0)
#print(x)

x.requires_grad_(True)
#print(x.grad)

y = 2 * torch.dot(x, x)
#print(y)

y.backward()
#print(x.grad)

#print(x.grad == 4 * x)

x.grad.zero_()
y = x.sum()
y.backward()
#print(x.grad)
```

```
# ============================== 2.5.2 ==================================

x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))
#print(x.grad)

# ============================== 2.5.3
==================================

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

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

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

# ============================== 2.5.4 ==============================

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()

#print(a.grad == d / a)

# as differentiation is a big part of training neural networs, having access
# to auto differentiation is a big time saver and helps minimise errors.

# ============================== 3.1
====================================
```

```
#============================== 3.1.2
=====================================

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]
#print(f'{time.time() - t:.5f} sec')

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


x = np.arange(-7, 7, 0.01)


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

#plt.show()

# linear regression is important when it comes to neural networks as
# it calculates the difference between the predicted outcome and actual
# outcome in a way that we can use to update weights and biases. it also
# emphasises the importance of using minibatches for training to increase
# the efficiency and gather results in an appropriate timeframe

# ================================ 3.2
===================================

# ================================ 3.2.1
===================================

def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
```

```
        setattr(Class, obj.__name__, obj)
    return wrapper

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

a = A()

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

#print(a.do())

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

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

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)

#plt.show()
```

```
# ================================ 3.2.2 ================================

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

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)


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

# object oriented design is always a useful way to make code reusable.
# in the case of a neural network we can easily make calls to class methods
# and run the network with a few lines in the main body which is efficient
```

```
# for space as well.

# ================================== 3.4.1
====================================

class LinearRegressionScratch(d2l.Module):
  """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)

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

# ================================== 3.4.2
====================================

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

# ================================== 3.4.3
==================================

class SGD(d2l.HyperParameters):
  """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_()

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

```
# ============================== 3.4.4
==================================

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

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

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}')

# this fully functioning neural network is a good base model to learn from
# and build upon when solving more complex problems.

# ================================ 4.1
======================================

# Entropy quantifies the uncertainty in a distribution, higher entropy means
# more unpredictability. cross entropy measures the average surprise when predicting
# and outcome using estimated probabilities, minimising this will imprve predictions.
# Softmax is a technique to convert outputs to probabilities between 0 and 1
```

```
# as well as ensuring probabilities arent negative and results in a classification being
made.
# it is efficient to represent categories as the probability of each one.
# for example {0, 1, 0} means the second category. this is one hot encoding.

# ================================ 4.2
====================================

# ================================ 4.2.1
======================================

d2l.use_svg_display()

class FashionMNIST(d2l.DataModule):
  """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)

data = FashionMNIST(resize=(32, 32))
#print(len(data.train), len(data.val), data.train[0][0].shape)

@d2l.add_to_class(FashionMNIST)
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
====================================

@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)

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

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
#print(f'{time.time() - tic:.2f} sec') #5.11 seconds

# ================================ 4.2.3
====================================

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

@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)

# using this fahsion dataset allows us to compare algorithms more effectively
# compared to the numbers dataset as these imiages may be more challenging to
# differentiate such as the pullover ns the shirt.

# ================================== 4.3
====================================

# ================================== 4.3.1
==================================

class Classifier(d2l.Module):
    """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)

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

# ================================== 4.3.2
==================================

@d2l.add_to_class(Classifier)
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

# this class gives us a base for classifiying and retrieving metrics from
# out neural network

# ================================== 4.4
================================

#================================== 4.4.1
================================

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

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

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

# ================================== 4.4.2
================================

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)
```

```
# ================================ 4.4.3
====================================

y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
#print(y_hat[[0, 1], y])

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

#print(cross_entropy(y_hat, y))

@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)

#plt.show()

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

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

# now the model is going through the steps and utilising softmax regression
# and updating weights and biases based on what causes lower entropy
# to make better predictions on the fashion-MNIST dataset

# ============================== 5.1
========================================

# ============================== 5.1.1
========================================

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))
#plt.show()
```

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

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

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))
#plt.show()

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

# this section shows how different activation functions are graphed and
# used for training neural networks using more than an input and output layer
# RELU signified a new standard for activation functions that allowed for more
# accurate updating of weights and biases

# ============================== 5.2
========================================

# ============================== 5.2.1
======================================
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)
#plt.show()

# ========================== 5.2.2
========================================

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)
#plt.show()

# here we added a hidden layer which allows the network to be more precise
# in which weights cause what outcome.

# ================================= 5.3
=========================================

# forward propagation and back prpagation are what the nural network uses to train
# on the dataset. forward propagation will take the input values, a vector relating
# to the data, and multiply it by the weights and add the biases all the way through
# to the output layer, which then gives a vector of probabilities of each category.
# backpropagation then uses the predicted probabilites and compares it to the actual
# probability to the go back and update the wights accordingly so that the data can
# be more correctly predicted if it went through the network again.
```