Preliminaries

Madiba Hudson-Quansah

Contents

[1]:
```
import torch
import numpy as np

np.random.seed(42)
torch.manual_seed(42)
```

[1]:
```
<torch._C.Generator at 0x700a702b22d0>
```

Chapter 1

Data Manipulation

1.1 Construction, Indexing, and Slicing

[2]:
```
x = torch.arange(12, dtype=torch.float32)
x
```

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

[3]:
```
x.numel()
```

[3]:
```
12
```

[4]:
```
X = x.reshape(3, -1)
X
```

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

[5]:
```
torch.zeros((2, 3, 4)), torch.ones((2, 3, 4))
```

[5]:
```
(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.]]]),
 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.]]]))
```

[6]:
```
torch.randn(3, 4)
```

[6]:
```
tensor([[ 0.3367,  0.1288,  0.2345,  0.2303],
        [-1.1229, -0.1863,  2.2082, -0.6380],
        [ 0.4617,  0.2674,  0.5349,  0.8094]])
```

[7]:
```
X[1, 3] = 17
X
```

[7]:
```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6., 17.],
        [ 8.,  9., 10., 11.]])
```

[8]:
```
X[0:2, 0:2] = 12
X
```

[8]:
```
tensor([[12., 12.,  2.,  3.],
        [12., 12.,  6., 17.],
        [ 8.,  9., 10., 11.]])
```

1.2 Operations and Broadcasting

[9]:
```
torch.exp(x)
```

[9]:
```
tensor([1.6275e+05, 1.6275e+05, 7.3891e+00, 2.0086e+01, 1.6275e+05, 1.
  ↪6275e+05,
        4.0343e+02, 2.4155e+07, 2.9810e+03, 8.1031e+03, 2.2026e+04, 5.
  ↪9874e+04])
```

[10]:
```
u = torch.tensor([1.0, 2, 4, 8])
v = torch.tensor([2, 2, 2, 2])
u + v, u - v, u * v, u / v, u**v
```

[10]:
```
(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.]))
```

[11]:
```
X = torch.arange(12, dtype=torch.float32).reshape(3, -1)
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)
```

[11]:
```
(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.]]))
```

[12]:
```
X == Y
```

[12]:
```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

[13]:
```
X.sum()
```

[13]:
```
tensor(66.)
```

[14]:
```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
a, b
```

[14]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

[15]:
```
a + b, a * b
```

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

1.3 Saving Memory

[16]:
```
before = id(Y)
Y[:] = X + Y
# or
# Y += X
after = id(Y)
f"{before:0x}", f"{after:0x}"
```

[16]:
```
('700982eeae90', '700982eeae90')
```

[17]:
```
Z = torch.zeros_like(Y)
print(f"{id(Z):0x}")
Z[:] = X + Y
print(f"{id(Z):0x}")
Z
```

```
700982ef54a0
700982ef54a0
```

[17]:
```
tensor([[ 2.,  3.,  8.,  9.],
        [ 9., 12., 15., 18.],
        [20., 21., 22., 23.]])
```

[18]:
```
A = X.numpy()
B = torch.from_numpy(A)
A, B
```

[18]:
```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]], dtype=float32),
 tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

[19]:
```
X > Y
```

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

[20]:
```
a = torch.round(100 * torch.randn((2, 3, 1)))
b = torch.round(100 * torch.randn((2, 1, 2)))
a, b
```

[20]:
```
(tensor([[[ 111.],
          [-169.],
          [ -99.]],

         [[  96.],
          [ 132.],
          [  82.]]]),
 tensor([[[-77., -75.]],

         [[135.,  69.]]]))
```

[21]:
```
a + b
```

[21]:
```
tensor([[[  34.,   36.],
         [-246., -244.],
         [-176., -174.]],

        [[ 231.,  165.],
         [ 267.,  201.],
         [ 217.,  151.]]])
```

Chapter 2

Data Preprocessing

[22]:
```
import os
import pandas as pd


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

data = pd.read_csv(dataFile)
data
```

[22]:

	NumRooms	RoofType	Price
0	NaN	NaN	127500
1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000

[23]:
```
inputs = data[data.columns[:-1]]
target = data[["Price"]]

inputs = pd.get_dummies(inputs, dummy_na=True)
inputs = inputs.fillna(inputs.mean())
inputs, target
```

[23]:

	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,

```
    Price
0  127500
1  106000
2  178100
3  140000)
```

[24]:
```
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(target.to_numpy(dtype=float))
X, y
```

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

Chapter 3

Linear Algebra

[25]:
```
A = torch.arange(6).reshape(3, -1)
A, A.T
```

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

[26]:
```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
A.T == A
```

[26]:
```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

[27]:
```
torch.arange(27).reshape(3, 3, 3)
```

[27]:
```
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],
         [24, 25, 26]]])
```

[28]:
```
A = torch.arange(6, dtype=torch.float32).reshape(2, -1)
B = A.clone()
A, A + B, A * B
```

[28]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
```

```
tensor([[ 0.,  2.,  4.],
        [ 6.,  8., 10.]]),
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]]))
```

[29]:
```
A.sum(), A.sum(axis=0), A.sum(axis=1)
```

[29]:
```
(tensor(15.), tensor([3., 5., 7.]), tensor([ 3., 12.]))
```

[30]:
```
A.mean(), A.sum() / A.numel(), A.sum() / (A.shape[0] * A.shape[1])
```

[30]:
```
(tensor(2.5000), tensor(2.5000), tensor(2.5000))
```

[31]:
```
A.sum(axis=1), A.sum(axis=1, keepdim=True), A.sum(axis=0), A.sum(axis=0,␣
↪keepdim=True)
```

[31]:
```
(tensor([ 3., 12.]),
 tensor([[ 3.],
         [12.]]),
 tensor([3., 5., 7.]),
 tensor([[3., 5., 7.]]))
```

[32]:
```
normMatrix = A / A.sum(axis=1, keepdim=True)
normMatrix, normMatrix.sum(axis=1, keepdim=True)
```

[32]:
```
(tensor([[0.0000, 0.3333, 0.6667],
         [0.2500, 0.3333, 0.4167]]),
 tensor([[1.],
         [1.]]))
```

[33]:
```
A, A.cumsum(axis=1)
```

[33]:
```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0.,  1.,  3.],
         [ 3.,  7., 12.]]))
```

[34]:
```
y = torch.ones(3, dtype=torch.float32)
x = torch.arange(3, dtype=torch.float32)
x, y, torch.dot(x, y), torch.sum(x * y)
```

[34]:
```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.), tensor(3.))
```

[35]:
```
u = torch.tensor([1, 2])
v = torch.tensor([3, 5])
diff = u - v
torch.sqrt(torch.dot(diff, diff))
```

[35]:
```
tensor(3.6056)
```

[36]:
```
A = torch.arange(9).reshape(3, -1)
x = torch.arange(3)
```

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

[36]:
```
(tensor([ 5, 14, 23]), tensor([ 5, 14, 23]))
```

[37]:
```
A = torch.randint(0, 30, (4, 4))
B = torch.randint(0, 30, (4, 4))
A, B, A @ B, torch.mm(A, B)
```

[37]:
```
(tensor([[27, 23, 23,  4],
         [ 3, 17,  0,  9],
         [10,  9, 16, 19],
         [25, 14,  8, 18]]),
 tensor([[26,  0, 20, 10],
         [ 0, 11,  3, 10],
         [ 1,  1, 17,  9],
         [ 4, 13, 28, 29]]),
 tensor([[ 741,  328, 1112,  823],
         [ 114,  304,  363,  461],
         [ 352,  362, 1031,  885],
         [ 730,  396, 1182,  984]]),
 tensor([[ 741,  328, 1112,  823],
         [ 114,  304,  363,  461],
         [ 352,  362, 1031,  885],
         [ 730,  396, 1182,  984]]))
```

[38]:
```
x = torch.randint(0, 30, (10,), dtype=float)
norm = torch.sqrt(x.dot(x))
x, norm, x.norm(), 2 * norm, torch.sqrt(2 * x.dot(2 * x)), (2 * x).norm()
```

[38]:
```
(tensor([ 3.,  7.,  8., 11., 24.,  1., 26.,  3., 22., 20.],
dtype=torch.float64),
 tensor(48.8774, dtype=torch.float64),
 tensor(48.8774, dtype=torch.float64),
 tensor(97.7548, dtype=torch.float64),
 tensor(97.7548, dtype=torch.float64),
 tensor(97.7548, dtype=torch.float64))
```

[39]:
```
x.abs().sum()
```

[39]:
```
tensor(125., dtype=torch.float64)
```

Generalized ℓ_p norm:

$$\|\mathbf{x}\|_p = (\sum_{i=1}^{n} |\ x_i\ |^p)^{1/p}$$

[40]:
```
def lpNorm(x: torch.Tensor, p=1):
    return torch.pow((torch.pow(x.abs(), p).sum()), (1 / p))
```

[41]:
```
lpNorm(x, 2)
```

[41]:
```
tensor(48.8774, dtype=torch.float64)
```

Frobenius norm:

$$\|X\| = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} x_{ij}^2}$$

[42]:
```
def frob(X: torch.Tensor):
    return torch.sqrt((X**2).sum())
```

[43]:
```
frob(X), torch.norm(X)
```

[43]:
```
(tensor(6.4807, dtype=torch.float64), tensor(6.4807, dtype=torch.float64))
```

[44]:
```
len(torch.randint(0, 10, (2, 3, 4)))
```

[44]:
```
2
```

[45]:
```
(A / A.sum(axis=0)).sum(axis=0)
```

[45]:
```
tensor([1., 1., 1., 1.])
```

[46]:
```
X = torch.randint(0, 30, (2, 3, 3), dtype=float)

X, torch.linalg.norm(X), torch.sqrt((X**2).sum())
```

[46]:
```
(tensor([[[15., 24., 21.],
          [21., 20., 29.],
          [20., 19., 21.]],

         [[18., 29., 16.],
          [17., 26., 10.],
          [ 9., 15.,  2.]]], dtype=torch.float64),
 tensor(83.1986, dtype=torch.float64),
 tensor(83.1986, dtype=torch.float64))
```

[47]:
```
A = torch.randint(0, 30, (100, 200), dtype=float)
B = torch.randint(0, 30, (100, 200), dtype=float)
C = torch.randint(0, 30, (100, 200), dtype=float)

Z = torch.stack((A, B, C), dim=0)
Z.shape
Z[1, :, :] == B
```

[47]:
```
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]])
```

Chapter 4

Calculus

[48]:
```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
import matplotlib.pyplot as plt
from matplotlib.scale import ScaleBase
from matplotlib.legend import Legend
from d2l import torch as d2l
```

[49]:
```
def f(x):
    return 3 * x**2 - 4 * x


def limit(f, x, h):
    return (f(x + h) - f(x)) / h


for h in 10.0 ** np.arange(-1, -10, -1):
    print(f"h={h:.5f}, numerical limit={limit(f, 1, h)}")
```

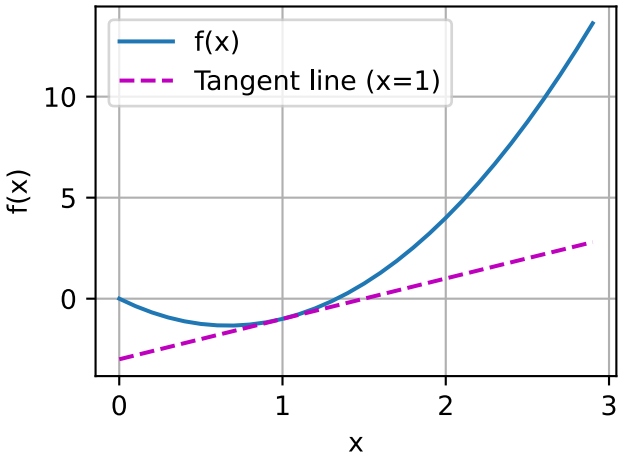
```
h=0.10000, numerical limit=2.3000000000000043
h=0.01000, numerical limit=2.0299999999999763
h=0.00100, numerical limit=2.002999999999311
h=0.00010, numerical limit=2.0002999999979565
h=0.00001, numerical limit=2.0000300000155846
h=0.00000, numerical limit=2.0000030001021685
h=0.00000, numerical limit=2.000000298707505
h=0.00000, numerical limit=1.999999987845058
h=0.00000, numerical limit=2.000000165480742
```

[50]:
```
from typing import Optional

type Limit = float | tuple[float, float]


def useSVGDisplay():
    backend_inline.set_matplotlib_formats("svg")
```

```python
def setFigSize(figsize=(3.5, 2.5)):
    useSVGDisplay()
    d2l.plt.rcParams["figure.figsize"] = figsize


def setAxes(
    axes: plt.Axes,
    xlabel: str,
    ylabel: str,
    xlim: Limit,
    ylim: Limit,
    xscale: ScaleBase,
    yscale: ScaleBase,
    legend: Optional[Legend] = None,
):
    axes.set_xlabel(xlabel), axes.set_ylabel(ylabel)
    axes.set_xscale(xscale), axes.set_yscale(yscale)
    axes.set_xlim(xlim), axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()


def plot(
    X,
    Y=None,
    xlabel: Optional[str] = None,
    ylabel: Optional[str] = None,
    legend: Optional[Legend] = [],
    xlim: Optional[Limit] = None,
    ylim: Optional[Limit] = None,
    xscale: ScaleBase = "linear",
    yscale: ScaleBase = "linear",
    fmts=("-", "m--", "g-.", "r:"),
    figsize=(3.5, 2.5),
    axes: plt.Axes = None,
):
    def hasOneAxis(X):
        return (
            hasattr(X, "ndim")
            and X.ndim == 1
            or isinstance(X, list)
            and not hasattr(X[0], "__len__")
        )

    if hasOneAxis(X):
        X = [X]

    if Y is None:
        X, Y = [[]] * len(X), X
    elif hasOneAxis(Y):
        Y = [Y]
```

```
    if len(X) != len(Y):
        X = X * len(Y)

    setFigSize(figsize)
    if axes is None:
        axes = d2l.plt.gca()
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        axes.plot(x, y, fmt) if len(x) else axes.plot(y, fmt)
    setAxes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

[51]:
```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], "x", "f(x)", legend=["f(x)", "Tangent line (x=1)"])
```



4.1 Partial Derivatives

Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The partial derivate of y with respect to its i^{th} parameter x_i is:

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1} + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

To calculate $\frac{\partial y}{\partial x_i}$, we treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative if y with respect to x_i. We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain a vector called the gradient of the function. Suppose that the input of function $f : \mathbb{R}^n \to \mathbb{R}$ is an n-dimensional vector $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\partial_{x_1} f(\mathbf{x}), \partial_{x_2} f(\mathbf{x}), \ldots, \partial_{x_n} f(\mathbf{x})\right]^T$$

When there is no ambiguity, $\nabla_{\mathbf{x}} f(\mathbf{x})$ can be replaced by $\nabla f(\mathbf{x})$. The following rules apply to differentiating multivariate functions: - For all $\mathbf{A} \in \mathbb{R}^{m \times n}$ we have $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^T$ and $\nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{A} = \mathbf{A}$ - For square matrices $\mathbf{A} \in \mathbb{R}^{n \times n}$ we have that $\nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^T)\mathbf{x}$ and in particular $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{x} = 2\mathbf{x}$

Similarly, for any matrix $\mathbf{X}$, we have $\nabla_{\mathbf{x}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$

4.2 Chain Rule

Suppose that $y = f(g(x))$ and that underlying functions $y = f(u)$ and $u = g(x)$, the chain rule states:

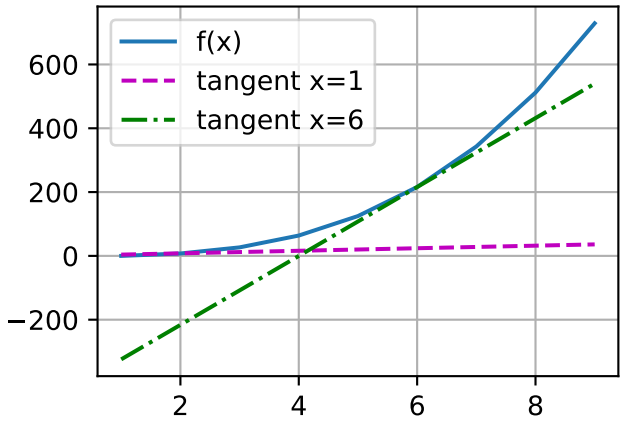
$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Generalizing to multivariate functions, suppose that $y = f(\mathbf{u})$ has variables $u_1, u_2, \ldots, u_m$, where each $u_i = g_i(\mathbf{x})$ has variables $x_1, x_2, \ldots, x_n$, i.e. $\mathbf{u} = g(\mathbf{x})$. Then the chain rule states that:

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1}\frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial x_i} + \ldots + \frac{\partial y}{\partial u_m}\frac{\partial u_m}{\partial x_i} \text{ and so } \nabla_{\mathbf{x}} y = \mathbf{A}\nabla_{\mathbf{u}} y$$

Where $\mathbf{A} \in \mathbb{n} \times \mathbb{m}$ is a matrix that contains the derivative of the vector $\mathbf{u}$ with respect to vector $\mathbf{v}$.

[52]:
```
x = np.arange(1, 10, 1)


def derv(x):
    return 3 * x**2 + x ** (-2)


def lin(x):
    m = derv(1)
    return x * m


def lin6(x):
    m = derv(6)
    return x * m - 1297 / 3


plot(
    x, [x**3 - (1 / x), lin(x), lin6(x)], legend=["f(x)", "tangent x=1",␣
 ↪"tangent x=6"]
)
```



Chapter 5

Automatic Differentiation (AutoGrad)

[53]:
```
x = torch.arange(4.0)
x
```

[53]:
```
tensor([0., 1., 2., 3.])
```

[54]:
```
x.requires_grad_(True)
x.grad
```

[55]:
```
y = 2 * x.dot(x)
y
```

[55]:
```
tensor(28., grad_fn=<MulBackward0>)
```

[56]:
```
y.backward()
x.grad
```

[56]:
```
tensor([ 0.,  4.,  8., 12.])
```

[57]:
```
x.grad == 4 * x
```

[57]:
```
tensor([True, True, True, True])
```

[58]:
```
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

[58]:
```
tensor([1., 1., 1., 1.])
```

[59]:
```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))
x, x.grad
```

[59]:
```
(tensor([0., 1., 2., 3.], requires_grad=True), tensor([0., 2., 4., 6.]))
```

[60]:
```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x
z
```

[60]:
```
tensor([ 0.,  1.,  8., 27.], grad_fn=<MulBackward0>)
```

[61]:
```
z.backward(gradient=torch.ones(len(y)))
x.grad == u, u, x.grad
```

[61]:
```
(tensor([True, True, True, True]),
 tensor([0., 1., 4., 9.]),
 tensor([0., 1., 4., 9.]))
```

[62]:
```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

[62]:
```
tensor([True, True, True, True])
```

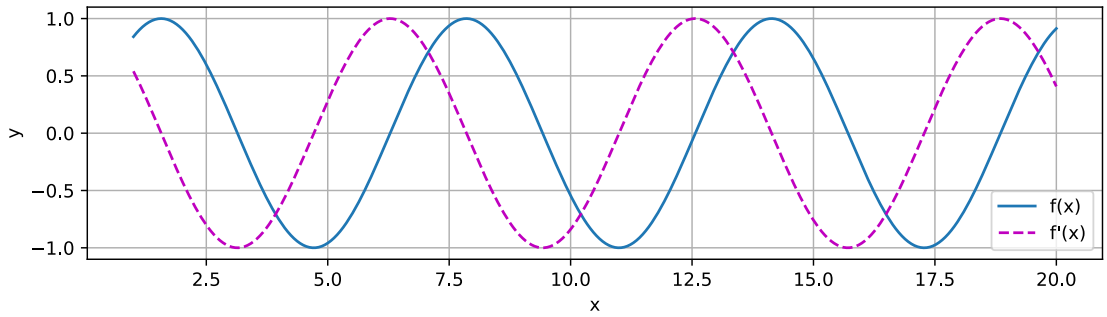
[63]:
```
def f(a: torch.Tensor):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

[64]:
```
a = torch.randn(size=(4,), requires_grad=True)
d = f(a)
d.backward(gradient=torch.ones_like(a))
```

[65]:
```
a.grad, a.grad == d / a
```

[65]:
```
(tensor([1024., 1024., 1024., 1024.]), tensor([True, True, True, True]))
```

[66]:
```
x = torch.linspace(1, 20, 300, requires_grad=True)
f = lambda x: torch.sin(x)
d = f(x)
e = f(x)
e.backward(gradient=torch.ones_like(x))
d = d.detach()
e = e.detach()
grad = x.grad
x = x.detach()
plot(x, [d, grad], "x", "y", ["f(x)", "f'(x)"], figsize=(10, 2.5))
```



$$a = \log x^2 b = \sin x c = a \times b d = x^-1 f(x) = c + d$$

```
graph TD;
    x["x"] -->|log x^2| a["a = log x^2"];
    x -->|sin x| b["b = sin x"];
    a --> c["c = a * b"];
    b --> c;
    x -->|x^-1| d["d = x^-1"];
    c --> f["f x = c + d"];
    d --> f;
```

Chapter 6

Probability and Statistics

[67]:
```
%matplotlib inline
import random
import torch
from torch.distributions.multinomial import Multinomial
from d2l import torch as d2l
```

[68]:
```
numToss = 100
heads = sum(random.random() > 0.5 for _ in range(numToss))
tails = numToss - heads
heads, tails
```

[68]:
```
(55, 45)
```

[69]:
```
coinProbs = torch.Tensor([0.5, 0.5])
Multinomial(numToss, coinProbs).sample() / numToss
```

[69]:
```
tensor([0.5000, 0.5000])
```

[70]:
```
numToss = 10000
Multinomial(numToss, coinProbs).sample() / numToss
```

[70]:
```
tensor([0.4890, 0.5110])
```

6.0.1 Law of Large Numbers

As the number of trials (n) increases, our estimates are guaranteed to converge to the true underlying probabilities.

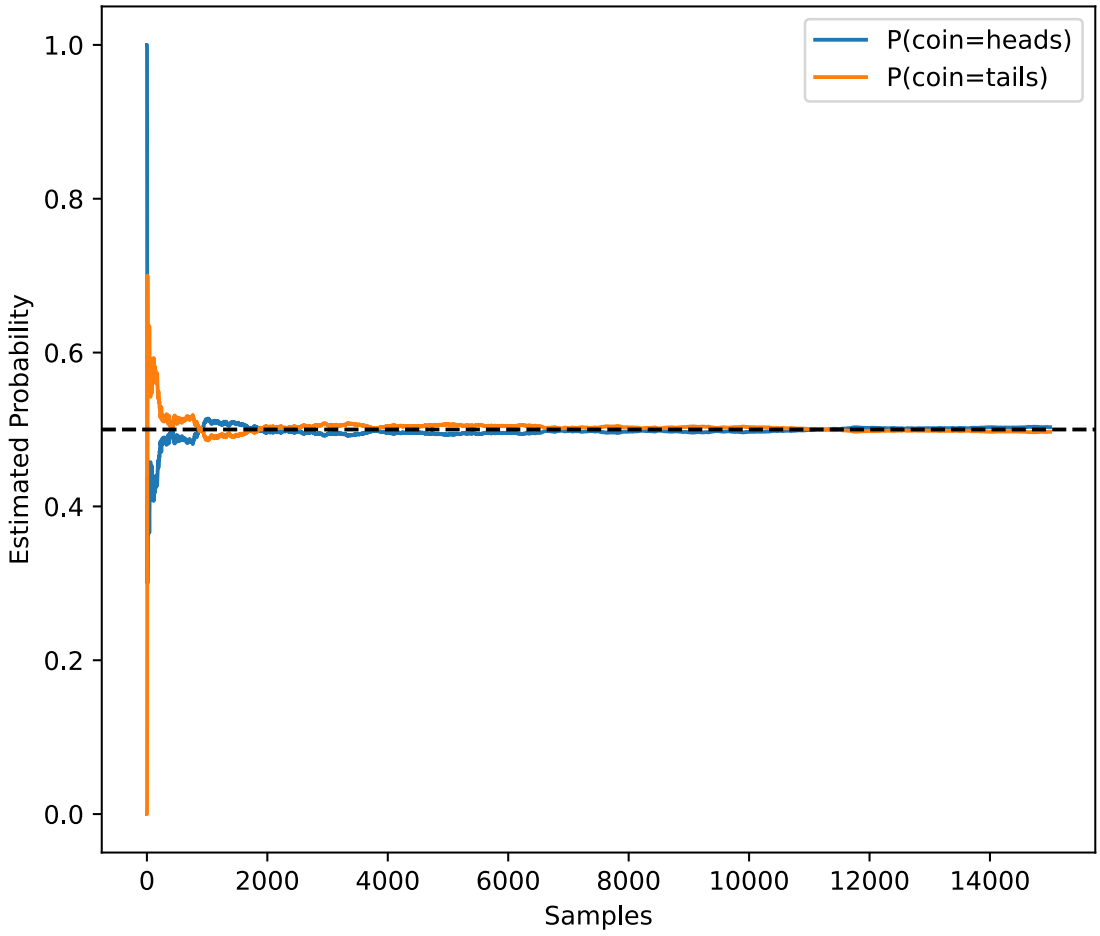
6.0.2 Central Limit Theorem

As the sample size n grows the absolute difference between the estimates and the underlying probabilities decrease at the rate of $\frac{1}{\sqrt{n}}$.

[71]:
```
counts = Multinomial(1, coinProbs).sample((15000,))
cumCounts = counts.cumsum(dim=0)
est = cumCounts / cumCounts.sum(dim=1, keepdims=True)
est = est.numpy()
```

```
d2l.set_figsize((7, 6))
d2l.plt.plot(est[:, 0], label=("P(coin=heads)"))
d2l.plt.plot(est[:, 1], label=("P(coin=tails)"))
d2l.plt.axhline(y=0.5, color="black", linestyle="dashed")
d2l.plt.gca().set_xlabel("Samples")
d2l.plt.gca().set_ylabel("Estimated Probability")
d2l.plt.legend()
```

[71]: <matplotlib.legend.Legend at 0x70095c8a1d10>



$\mathcal{S}$ denotes the sample/outcome space, i.e. the set of all possible outcomes. In the case of rolling a six sided die, $\mathcal{S} = 1, 2, 3, 4, 5, 6$. Events are subsets of the sample space for example rolling a dice and getting a 3 is an event and 3 is a subset of the sample space $\mathcal{S}$. Whenever the outcome z of a random experiment satisfies $z \in \mathcal{A}$ then event $\mathcal{A}$ has occurred, for example we could define the events "seeing a 5" ($\mathcal{A} = 5$) and "seeing an odd number" ($\mathcal{B} = 1, 3, 5$). In the case $z = 5$ we can say both events $\mathcal{A}$ and $\mathcal{B}$ occurred.

A probability function maps events to real values

$$\mathcal{P} \subseteq \mathcal{S} \rightarrow [0, 1]$$

The probability denoted by $\mathcal{P}(\mathcal{A})$ of an event $\mathcal{A}$ happening within a particular sample space has the following

properties - The probability of any event $\mathcal{A}$ is a nonnegative number: $\mathcal{P}(\mathcal{A}) \geqslant 0$ - The probability of the entire sample space is 1: $\mathcal{P}(\mathcal{S}) = 1$ - For any countable sequence of events $\mathcal{A}_\infty, \mathcal{A}_\in, \ldots$ that are mutually exclusive the probability that any of them happens is equal to the sum of their probabilities : $\mathcal{P}(\mathcal{A}_\infty \cup \mathcal{A}_\in \cup \ldots \cup \mathcal{A}_\setminus) = \sum_{i=i}^{n} \mathcal{P}(\mathcal{A}_\rangle)$

A random variable is defined by some relationship to the sample space, for example the number of heads in a series of coin flips, denoted by X. The occurrence where the random variable X takes the value of v is denoted as $X = v$ and the probability of this event is thus $P(X = v)$

Multiple random variables can interact with each other, i.e. influencing the probability of each other taking a certain value in the sample space. This is denoted as $P(X, Y) = P(X)P(Y)$ for independent random variables X and Y. We can thus construct every combination of values that the variables can jointly take, this is called a joint probability function. This function returns the assigned probability to the intersection of the corresponding subsets of the sample space. This joint probability assigned to the event where the random variables X and Y take the values x and y respectively is denoted as $P(X = x, Y = y)$, where the comma indicates "and", it then follows that

$$P(X = x, Y = y) \leqslant P(X = x) \text{ and } P(X = x, Y = y) \leqslant P(Y = y)$$

since for $X = x$ and $Y = y$ to occur, $X = x$ and $Y = y$ must occur. This joint probability can be used to derive a lot of information about the random variables X and Y including their probability distributions $P(X)$ and $P(Y)$ respectively. To derive $P(X = x)$ we can sum over all the probabilities of $P(X = x, Y = v)$ where v is all the values Y can take determined by the sample space, i.e.:

$$P(X = x) = \sum_v P(X = x, Y = v)$$

The ratio

$$\frac{P(X = x, Y = y)}{P(Y = y)} \leqslant 1$$

Is denoted as the conditional probability denoted with the | symbol:

$$P(Y = y | X = x) = \frac{P(X = x, Y = y)}{P(Y = y)}$$

And can be used to determine the probability of $Y = y$ knowing that $X = x$ has occurred. This can be thought of as restricting out focus to only the subset of the sample space associated with $X = x$ and then renormalizing so that all probabilities of this subset sum to 1.

Using this definition of conditional probability we can derive Bayes' Theorem. By construction we have that $P(X, Y) = P(Y|X)P(X)$ and $P(X, Y) = P(X|Y)P(Y)$. Hence:

$$P(Y|X)P(X) = P(X|Y)P(Y)P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$$

In some cases we might not have direct access to $P(Y)$ this is where the simplified Bayes' Theorem comes in:

$$P(X|Y) \propto P(Y|X)P(X)$$

Since we know $P(X|Y)$ must be normalized i.e. $\sum_x P(X = x|Y = y) = 1$, i.e.:

$$P(A|B) = \frac{P(Y|X)P(X)}{\sum_x P(Y|X = x)P(X = x)}$$

In Bayesian statistics an observer is believed to possess some prior beliefs about the the plausibility of the available hypotheses encoded in the prior probability $P(H)$, and a likelihood function that determines how likely one is to observe any value of the collected evidence for each of the hypotheses in the class $P(E \mid H)$. Using this logic Bayes' theorem can be adapted to inform how to update the prior beliefs $P(H)$ in light of new evidence $P(E)$ to produce posterior beliefs $P(H \mid E)$:

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)}$$

I.e. The posterior equals prior times the likelihood divided by the evidence.