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([[-0.9835, -0.2872, -0.6260,  0.4682],
        [ 0.2183,  0.8327,  0.5552,  1.2401],
        [ 1.0363, -0.9552,  1.8169, -1.0028]])
```

```
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): 133819837977984
id(Z): 133819837977984
```

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

```
True
```

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

```
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)
```

더블클릭 또는 Enter 키를 눌러 수정

Discussion and Exercise

```
# Discussion
'''
딥러닝의 모든 시작은 텐서에서 되는만큼, 텐서의 구조와 그 명령어를 정확히 이해하고 있을 필요가 있다.
 딥러닝 연산 과정에서 텐서의 차원이 맞지 않는 경우 오류가 발생하곤 하는데, 텐서의 구조를 정확히 이해하고 있지 않으면 그때마다 상당한 곤란함을 겪게 될
'''
```

```
# Exercise 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)
```

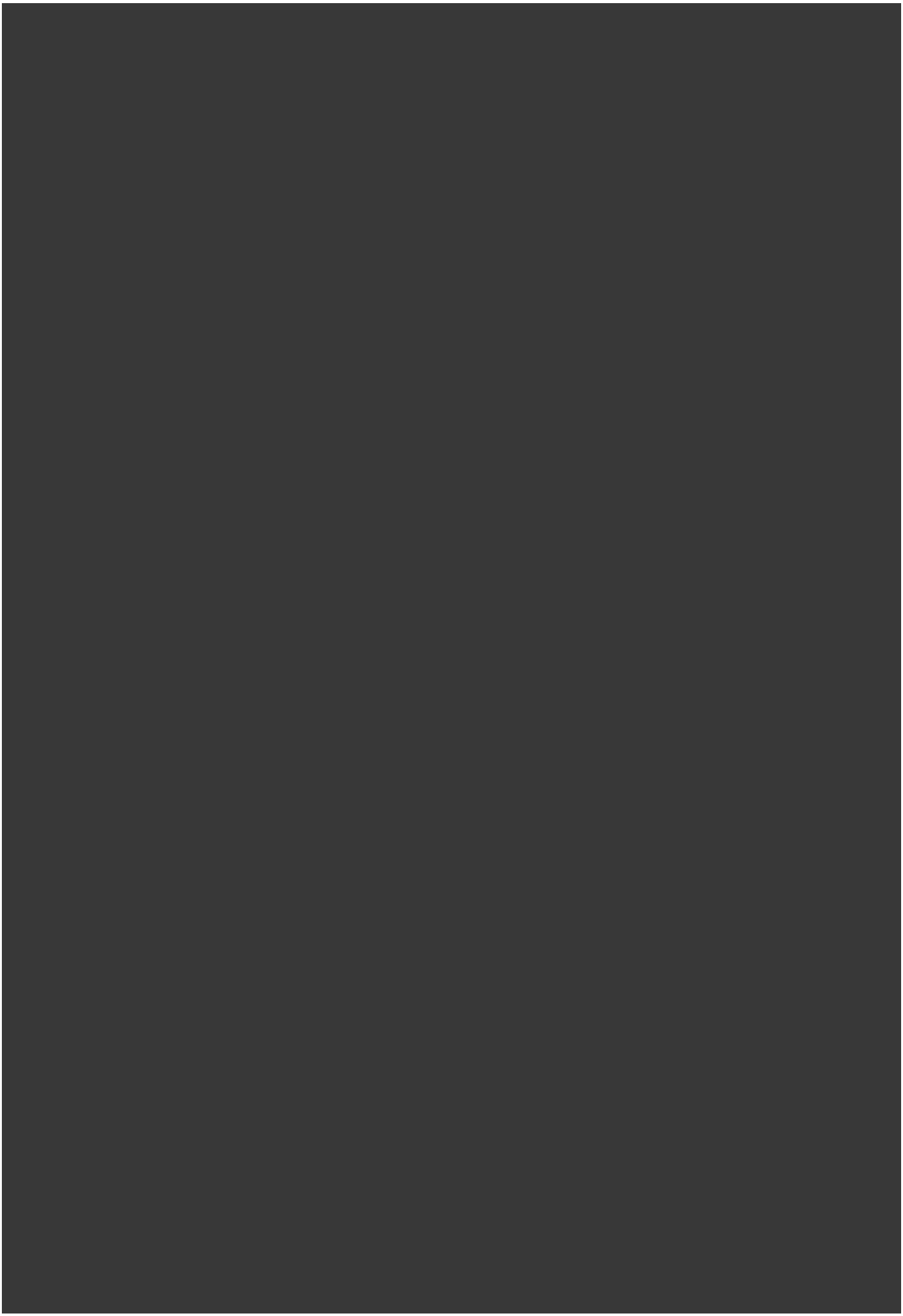
```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
      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

```
print(X == Y) # original
print(X < Y)
print(X > Y)
```

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

```
# Exercise 2

a = torch.arange(6).reshape((3, 2, 1))
b = torch.arange(2).reshape((1, 2))
print(a)
print(b)
print('Wn##### RESULT #####Wn')
print(a + b)
```

```
tensor([[[0],
         [1]],

        [[2],
         [3]],

        [[4],
         [5]]])
tensor([[0, 1]])

##### RESULT #####

tensor([[[0, 1],
         [1, 2]],

        [[2, 3],
         [3, 4]],

        [[4, 5],
         [5, 6]]])
```



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               0             1
1       2.0               0             1
2       4.0               1             0
3       NaN               0             1
```

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

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

```
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))
```

Discussion

```
'''
- 데이터 전처리 과정이 생각보다 중요하다.
'''
```

You now know how to partition data columns, impute missing variables, and load `pandas` data into tensors. In :numref:`sec_kaggle_house`, you will pick up some more data processing skills. While this crash course kept things simple, data processing can get hairy. For example, rather than arriving in a single CSV file, our dataset might be spread across multiple files extracted from a relational database. For instance, in an e-commerce application, customer addresses might live in one table and purchase data in another. Moreover, practitioners face myriad data types beyond categorical and numeric, for example, text strings, images, audio data, and point clouds. Oftentimes, advanced tools and efficient algorithms are required in order to prevent data processing from becoming the biggest bottleneck in the machine learning pipeline. These problems will arise when we get to computer vision and natural language processing. Finally, we must pay attention to data quality. Real-world datasets are often plagued by outliers, faulty measurements from sensors, and recording errors, which must be addressed before feeding the data into any model. Data visualization tools such as seaborn, Bokeh, or matplotlib can help you to manually inspect the data and develop intuitions about the type of problems you may need to address.

Exercises

1. Try loading datasets, e.g., Abalone from the UCI Machine Learning Repository and inspect their properties. What fraction of them has missing values? What fraction of the variables is numerical, categorical, or text?
2. Try indexing and selecting data columns by name rather than by column number. The pandas documentation on indexing has further details on how to do this.
3. How large a dataset do you think you could load this way? What might be the limitations? Hint: consider the time to read the data, representation, processing, and memory footprint. Try this out on your laptop. What happens if you try it out on a server?
4. How would you deal with data that has a very large number of categories? What if the category labels are all unique? Should you include the latter?
5. What alternatives to pandas can you think of? How about loading NumPy tensors from a file? Check out Pillow, the Python Imaging Library.

```
# Exercise 1
import sklearn
from sklearn.datasets import load_iris

iris = load_iris()
iris_data = pd.DataFrame(iris.data, columns=iris.feature_names)
iris_data["target"] = iris.target
iris_data.head() # 데이터 확인
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

```
# 데이터셋 기본정보 확인
iris_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   target             150 non-null    int64  
dtypes: float64(4), int64(1)
memory usage: 6.0 KB
```

```
# 데이터셋의 각 열별 결측치 개수(여부) 확인
iris_data.isnull().sum()
```

```
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
target               0
dtype: int64
```

```
# Exercise 2
column_name = 'sepal length (cm)'
selected_column = iris_data[column_name]
print(selected_column)
```

```
0      5.1
1      4.9
2      4.7
3      4.6
4      5.0
      ...
145    6.7
146    6.3
147    6.5
148    6.2
149    5.9
Name: sepal length (cm), Length: 150, dtype: float64
```

```
# Exercise 3
print('데이터를 로드하는 데 걸리는 시간 혹은 코스트에는 여러가지 요소가 영향을 미칠 수 있다. 다음은 그 예시이다. 하드웨어나 메모리 용량, 데이터셋 구조
print('실제로 GB 단위의 매우 큰 데이터셋을 사용하고자 할 때, 컴퓨터 시스템 자원의 부족으로 데이터 로드조차 안되는 경우가 종종 있다.')
```

```
데이터를 로드하는 데 걸리는 시간 혹은 코스트에는 여러가지 요소가 영향을 미칠 수 있다. 다음은 그 예시이다. 하드웨어나 메모리 용량, 데이터셋 구조
실제로 GB 단위의 매우 큰 데이터셋을 사용하고자 할 때, 컴퓨터 시스템 자원의 부족으로 데이터 로드조차 안되는 경우가 종종 있다.
```

```
# Exercise 4
print('Q: How would you deal with data that has a very large number of categories?')
print('A: feature engineering 을 통해 category 의 개수를 줄임으로써, 데이터의 차원을 줄인다.')
print()
print('Q: What if the category labels are all unique? ')
print('A: 주어진 labe 들을 유사도 등을 기준으로, 혹은 특정 알고리즘을 기반으로 클러스터링하여 비슷한 특성의 것들끼리 묶고, 같은 클러스터 끼리는 모두 하
```

```
Q: How would you deal with data that has a very large number of categories?
A: feature engineering 을 통해 category 의 개수를 줄임으로써, 데이터의 차원을 줄인다.

Q: What if the category labels are all unique?
A: 주어진 labe 들을 유사도 등을 기준으로, 혹은 특정 알고리즘을 기반으로 클러스터링하여 비슷한 특성의 것들끼리 묶고, 같은 클러스터 끼리는 모두 하
```

```
# Exercise 5
print('Numpy, Dask, Vaex 등의 alternative 가 존재한다.')
```

```
Numpy, Dask, Vaex 등의 alternative 가 존재한다.
```

Linear Algebra

```
import torch
```

Scalars

```
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.))
```

Vectors

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

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

```
x[2]
```

```
tensor(2)
```

```
len(x)
```

```
3
```

```
x.shape
```

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

Matrices

```
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]])
```

Tensors

```
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]]])
```

Basic Properties of Tensor Arithmetic

```
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]))
```

Reduction

```
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]))
```

Non-Reduction Sum

```
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.]])
```

Dot Products

```
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.)
```

Matrix--Vector Products

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

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

Matrix--Matrix Multiplication

```
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.]]))
```

Norms

```
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.)
```

Discussion (TakeAway Msg)

```
'''
- axis 옵션을 통해 sum 을 여러가지 방식으로 할 수 있다.
- dot product (내적) 와 element-wise produc 의 코드 상 차이를 명확히 알 필요가 있다.
'''
```

In this section, we have reviewed all the linear algebra that you will need to understand a significant chunk of modern deep learning. There is a lot more to linear algebra, though, and much of it is useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you will be more inclined to learn more mathematics once you have gotten your hands dirty applying machine learning to real datasets. So while we reserve the right to introduce more mathematics later on, we wrap up this section here.

If you are eager to learn more linear algebra, there are many excellent books and online resources. For a more advanced crash course, consider checking out :citet:`Strang.1993`, :citet:`Kolter.2008`, and :citet:`Petersen.Pedersen.ea.2008`.

To recap:

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as `sum` and `mean`, respectively.
- Elementwise products are called Hadamard products. By contrast, dot products, matrix–vector products, and matrix–matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix–matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the ℓ_1 and ℓ_2 norms, and common matrix norms include the *spectral* and *Frobenius* norms.

Automatic Differentiation

```
import torch
```

A Simple Function

```
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.])
```

Backward for Non-Scalar Variables

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

```
tensor([0., 2., 4., 6.])
```

Detaching Computation

```
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])
```

Gradients and Python Control Flow

```
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)
```

Discussion

```
'''
자코비안(Jacobian) 행렬이란 단순히 '편미분' 자체를 의미하는 것이 아니라, 모든 벡터들의 1차 편미분된 값들로 이루어진 행렬을 의미한다.
다시 말해, 각 행렬의 값은 다변수 함일 때의 미분값이다.

.backward() 를 호출하면 역전파 과정이 시작되는 것이다.
이 과정을 통해, 매개변수가 조정된다.

참고자료 https://tutorials.pytorch.kr/beginner/blitz/autograd_tutorial.html
'''
```

You have now gotten a taste of the power of automatic differentiation. The development of libraries for calculating derivatives both automatically and efficiently has been a massive productivity booster for deep learning practitioners, liberating them so they can focus on less menial. Moreover, autograd lets us design massive models for which pen and paper gradient computations would be prohibitively time consuming. Interestingly, while we use autograd to *optimize* models (in a statistical sense) the *optimization* of autograd libraries themselves (in a computational sense) is a rich subject of vital interest to framework designers. Here, tools from compilers and graph manipulation are leveraged to compute results in the most expedient and memory-efficient manner.

For now, try to remember these basics: (i) attach gradients to those variables with respect to which we desire derivatives; (ii) record the computation of the target value; (iii) execute the backpropagation function; and (iv) access the resulting gradient.

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

Linear Regression

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

Basics

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

+ 코드 + 텍스트

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

```
'0.22844 sec'
```

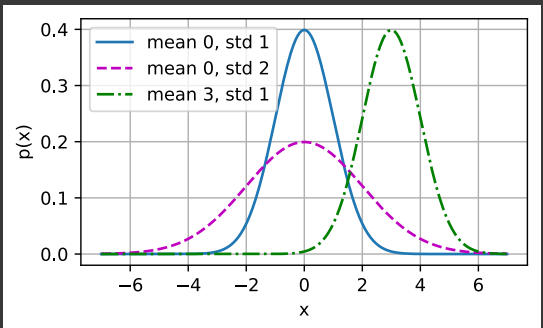
```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
'0.00026 sec'
```

The Normal Distribution and Squared Loss

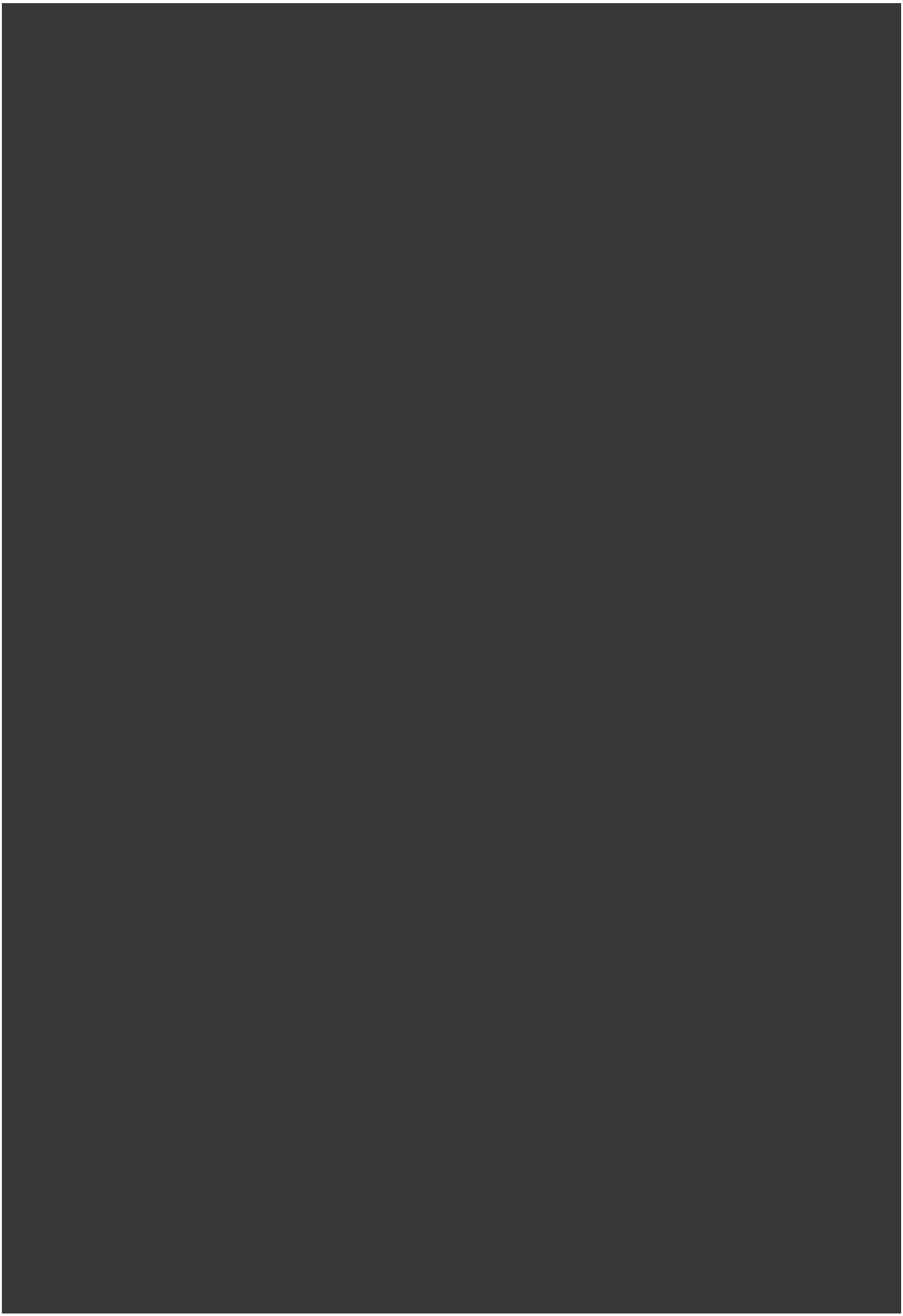
```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

```
# 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])
```



Discussion

```
'''
- 선형회귀는 선형 함수 (예측 함수) 와 실제 데이터포인트 사이의 L1 Loss 또는 L2 Loss 를 최소화하는 방향으로 학습된다.
- element-wise operation 이 단순 loop 문 보다 훨씬 빠르다.
'''
```



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

Object-Oriented Design for Implementation

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

Utilities

```
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)

a.do()
```

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

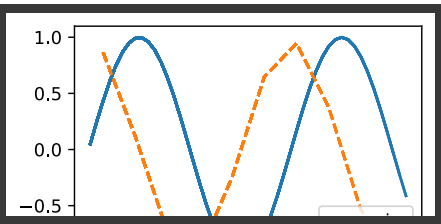
```
class HyperParameters:
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

```
# 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)
```

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

```
class ProgressBoard(d2l.HyperParameters):
    """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)
```



Models

```
class Module(nn.Module, d2l.HyperParameters):
    """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
```

Data

```
class DataModule(d2l.HyperParameters):
    """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)
```

Training

```
class Trainer(d2l.HyperParameters):
    """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
```

Discussion

```
'''
- 딥러닝 모델의 구현을 위해서는 여러가지 파이썬 클래스 정의가 선행되어야 한다.
이때 필요한 클래스는 다음과 같다:
1. Model 클래스
2. Data 클래스
3. Trian 클래스 (실제 코드들 보면, Train 은 굳이 클래스로 만들지 않는 경우도 많다.)
'''
```

To highlight the object-oriented design for our future deep learning implementation, the above classes simply show how their objects store data and interact with each other. We will keep enriching implementations of these classes, such as via `@add_to_class`, in the rest of the book. Moreover, these fully implemented classes are saved in the D2L library, a *lightweight toolkit* that makes structured modeling for deep learning easy. In particular, it facilitates reusing many components between projects without changing much at all. For instance, we can replace just the optimizer, just the model, just the dataset, etc.; this degree of modularity pays dividends throughout the book in terms of conciseness and simplicity (this is why we added it) and it can do the same for your own projects.

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

Linear Regression Implementation from Scratch

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

Defining the Model

```
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
```

Defining the Loss Function

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

Defining the Optimization Algorithm

```
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)
```

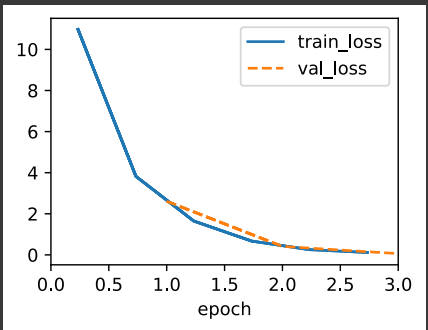
Training

```
@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:  # 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
```

```
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([ 0.1142, -0.2129])
error in estimating b: tensor([0.2531])
```

Discussion

```
'''
- .matmul 은 matrix-vector product 를 할 때 사용한다. (일반적으로 input features  X  and the model weights  w)
- weights 를 초기화할때 일반적으로 정규분포를 따르는 랜덤한 값으로 초기화시킨다. 이때 평균 0, 표준편차 0.01로 설정하는 것이 magic number 라고 한다.
- 사실 요즘 대부분의 메소드들이 사용하기 편한 high-level 의 형태로 구현되어 있지만, 모델을 커스텀하기 위해서는 결국 on the scratch 로 모델을 구현해보는
'''
```

Discussion (TakeAway Msg)

```
'''
- Softmax 함수는 ouput 으로서 일종의 확률분포 값을 내놓음으로써,discrete 한 output space 를 최적화할 때 사용한다.
- 따라서, 분류 태스크에서 주로 사용되곤 한다.
'''
```

Softmax Regression

:label:`sec_softmax`

In :numref:`sec_linear_regression`, we introduced linear regression, working through implementations from scratch in :numref:`sec_linear_scratch` and again using high-level APIs of a deep learning framework in :numref:`sec_linear_concise` to do the heavy lifting.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (price) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you are probably looking for a regression model. However, even within regression models, there are important distinctions. For instance, the price of a house will never be negative and changes might often be *relative* to its baseline price. As such, it might be more effective to regress on the logarithm of the price. Likewise, the number of days a patient spends in hospital is a *discrete nonnegative* random variable. As such, least mean squares might not be an ideal approach either. This sort of time-to-event modeling comes with a host of other complications that are dealt with in a specialized subfield called *survival modeling*.

The point here is not to overwhelm you but just to let you know that there is a lot more to estimation than simply minimizing squared errors. And more broadly, there is a lot more to supervised learning than regression. In this section, we focus on *classification* problems where we put aside *how much?* questions and instead focus on *which category?* questions.

- Does this email belong in the spam folder or the inbox?
- Is this customer more likely to sign up or not to sign up for a subscription service?
- Does this image depict a donkey, a dog, a cat, or a rooster?
- Which movie is Aston most likely to watch next?
- Which section of the book are you going to read next?

Colloquially, machine learning practitioners overload the word *classification* to describe two subtly different problems: (i) those where we are interested only in hard assignments of examples to categories (classes); and (ii) those where we wish to make soft assignments, i.e., to assess the probability that each category applies. The distinction tends to get blurred, in part, because often, even when we only care about hard assignments, we still use models that make soft assignments.

Even more, there are cases where more than one label might be true. For instance, a news article might simultaneously cover the topics of entertainment, business, and space flight, but not the topics of medicine or sports. Thus, categorizing it into one of the above categories on their own would not be very useful. This problem is commonly known as multi-label classification. See :citet:`Tsoumakas.Katakis.2007` for an overview and :citet:`Huang.Xu.Yu.2015` for an effective algorithm when tagging images.

Classification

:label:`subsec_classification-problem`

To get our feet wet, let's start with a simple image classification problem. Here, each input consists of a 2×2 grayscale image. We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4. Further, let's assume that each image belongs to one among the categories "cat", "chicken", and "dog".

Next, we have to choose how to represent the labels. We have two obvious choices. Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent $\{\text{dog}, \text{cat}, \text{chicken}\}$ respectively. This is a great way of *storing* such information on a computer. If the categories had some natural ordering among them, say if we were trying to predict $\{\text{baby}, \text{toddler}, \text{adolescent}, \text{young adult}, \text{adult}, \text{geriatric}\}$, then it might even make sense to cast this as an ordinal regression problem and keep the labels in this format. See :citet:`Moon.Smola.Chang.ea.2010` for an overview of different types of ranking loss functions and :citet:`Beutel.Murray.Faloutsos.ea.2014` for a Bayesian approach that addresses responses with more than one mode.

In general, classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one-hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to a particular instance's category is set to 1 and all other components are set to 0. In our case, a label y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken", and $(0, 0, 1)$ to "dog":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

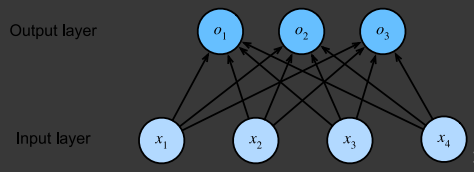
Linear Model

In order to estimate the conditional probabilities associated with all the possible classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many affine functions as we have outputs. Strictly speaking, we only need one

fewer, since the final category has to be the difference between 1 and the sum of the other categories, but for reasons of symmetry we use a slightly redundant parametrization. Each output corresponds to its own affine function. In our case, since we have 4 features and 3 possible output categories, we need 12 scalars to represent the weights (w with subscripts), and 3 scalars to represent the biases (b with subscripts). This yields:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned}$$

The corresponding neural network diagram is shown in :numref:`fig_softmaxreg`. Just as in linear regression, we use a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on every input, x_1, x_2, x_3, and x_4, the output layer can also be described as a *fully connected layer*.



:label:`fig_softmaxreg`

For a more concise notation we use vectors and matrices: $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$ is much better suited for mathematics and code. Note that we have gathered all of our weights into a 3×4 matrix and all biases $\mathbf{b} \in \mathbb{R}^3$ in a vector.

The Softmax

:label:`subsec_softmax_operation`

Assuming a suitable loss function, we could try, directly, to minimize the difference between $\mathbf{o}$ and the labels $\mathbf{y}$. While it turns out that treating classification as a vector-valued regression problem works surprisingly well, it is nonetheless unsatisfactory in the following ways:

- There is no guarantee that the outputs o_i sum up to 1 in the way we expect probabilities to behave.
- There is no guarantee that the outputs o_i are even nonnegative, even if their outputs sum up to 1, or that they do not exceed 1.

Both aspects render the estimation problem difficult to solve and the solution very brittle to outliers. For instance, if we assume that there is a positive linear dependency between the number of bedrooms and the likelihood that someone will buy a house, the probability might exceed 1 when it comes to buying a mansion! As such, we need a mechanism to "squish" the outputs.

There are many ways we might accomplish this goal. For instance, we could assume that the outputs $\mathbf{o}$ are corrupted versions of $\mathbf{y}$, where the corruption occurs by means of adding noise $\boldsymbol{\epsilon}$ drawn from a normal distribution. In other words, $\mathbf{y} = \mathbf{o} + \boldsymbol{\epsilon}$, where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. This is the so-called [probit model], first introduced by :citet:`Fechner.1860`. While appealing, it does not work quite as well nor lead to a particularly nice optimization problem, when compared to the softmax.

Another way to accomplish this goal (and to ensure nonnegativity) is to use an exponential function $P(y = i) \propto \exp o_i$. This does indeed satisfy the requirement that the conditional class probability increases with increasing o_i, it is monotonic, and all probabilities are nonnegative. We can then transform these values so that they add up to 1 by dividing each by their sum. This process is called *normalization*. Putting these two pieces together gives us the *softmax* function:

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}.$$

:eqlabel:`eq_softmax_y_and_o`

Note that the largest coordinate of $\mathbf{o}$ corresponds to the most likely class according to $\hat{\mathbf{y}}$. Moreover, because the softmax operation preserves the ordering among its arguments, we do not need to compute the softmax to determine which class has been assigned the highest probability. Thus,

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j.$$

The idea of a softmax dates back to :citet:`Gibbs.1902`, who adapted ideas from physics. Dating even further back, Boltzmann, the father of modern statistical physics, used this trick to model a distribution over energy states in gas molecules. In particular, he discovered that the prevalence of a state of energy in a thermodynamic ensemble, such as the molecules in a gas, is proportional to $\exp(-E/kT)$. Here, E is the energy of a state, T is the temperature, and k is the Boltzmann constant. When statisticians talk about increasing or decreasing the "temperature" of a statistical system, they refer to changing T in order to favor lower or higher energy states. Following Gibbs' idea, energy equates to error. Energy-based models :cite:`Ranzato.Boureau.Chopra.ea.2007` use this point of view when describing problems in deep learning.

Vectorization

:label:`subsec_softmax_vectorization`

To improve computational efficiency, we vectorize calculations in minibatches of data. Assume that we are given a minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ of n examples with dimensionality (number of inputs) d. Moreover, assume that we have q categories in the output. Then the weights satisfy $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned}$$

:eqlabel: `eq_minibatch_softmax_reg`

This accelerates the dominant operation into a matrix--matrix product $\mathbf{XW}$. Moreover, since each row in $\mathbf{X}$ represents a data example, the softmax operation itself can be computed *rowwise*: for each row of $\mathbf{O}$, exponentiate all entries and then normalize them by the sum. Note, though, that care must be taken to avoid exponentiating and taking logarithms of large numbers, since this can cause numerical overflow or underflow. Deep learning frameworks take care of this automatically.

Loss Function

:label: `subsec_softmax-regression-loss-func`

Now that we have a mapping from features $\mathbf{x}$ to probabilities $\hat{\mathbf{y}}$, we need a way to optimize the accuracy of this mapping. We will rely on maximum likelihood estimation, the very same method that we encountered when providing a probabilistic justification for the mean squared error loss in :numref: `subsec_normal_distribution_and_squared_loss`.

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as the (estimated) conditional probabilities of each class, given any input $\mathbf{x}$, such as $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$. In the following we assume that for a dataset with features $\mathbf{X}$ the labels $\mathbf{Y}$ are represented using a one-hot encoding label vector. We can compare the estimates with reality by checking how probable the actual classes are according to our model, given the features:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}).$$

We are allowed to use the factorization since we assume that each label is drawn independently from its respective distribution $P(\mathbf{y} \mid \mathbf{x}^{(i)})$. Since maximizing the product of terms is awkward, we take the negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

where for any pair of label $\mathbf{y}$ and model prediction $\hat{\mathbf{y}}$ over q classes, the loss function l is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j.$$

:eqlabel: `eq_l_cross_entropy`

For reasons explained later on, the loss function in :eqref: `eq_l_cross_entropy` is commonly called the *cross-entropy loss*. Since $\mathbf{y}$ is a one-hot vector of length q, the sum over all its coordinates j vanishes for all but one term. Note that the loss $l(\mathbf{y}, \hat{\mathbf{y}})$ is bounded from below by 0 whenever $\hat{\mathbf{y}}$ is a probability vector: no single entry is larger than 1, hence their negative logarithm cannot be lower than 0; $l(\mathbf{y}, \hat{\mathbf{y}}) = 0$ only if we predict the actual label with *certainty*. This can never happen for any finite setting of the weights because taking a softmax output towards 1 requires taking the corresponding input o_i to infinity (or all other outputs o_j for $j \neq i$ to negative infinity). Even if our model could assign an output probability of 0, any error made when assigning such high confidence would incur infinite loss ($-\log 0 = \infty$).

Softmax and Cross-Entropy Loss

:label: `subsec_softmax_and_derivatives`

Since the softmax function and the corresponding cross-entropy loss are so common, it is worth understanding a bit better how they are computed. Plugging :eqref: `eq_softmax_y_and_o` into the definition of the loss in :eqref: `eq_l_cross_entropy` and using the definition of the softmax we obtain

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned}$$

To understand a bit better what is going on, consider the derivative with respect to any logit o_j. We get

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j.$$

In other words, the derivative is the difference between the probability assigned by our model, as expressed by the softmax operation, and what actually happened, as expressed by elements in the one-hot label vector. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate $\hat{y}$. This is not a coincidence. In any exponential family model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Now consider the case where we observe not just a single outcome but an entire distribution over outcomes. We can use the same representation as before for the label $\mathbf{y}$. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l in :eqref:`eq_l_cross_entropy` still works well, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels. This loss is called the *cross-entropy loss* and it is one of the most commonly used losses for classification problems. We can demystify the name by introducing just the basics of information theory. In a nutshell, it measures the number of bits needed to encode what we see, $\mathbf{y}$, relative to what we predict that should happen, $\hat{\mathbf{y}}$. We provide a very basic explanation in the following. For further details on information theory see :citet:`Cover.Thomas.1999` or :citet:`mackay2003information`.

Information Theory Basics

:label:`subsec_info_theory_basics`

Many deep learning papers use intuition and terms from information theory. To make sense of them, we need some common language. This is a survival guide. *Information theory* deals with the problem of encoding, decoding, transmitting, and manipulating information (also known as data).

Entropy

The central idea in information theory is to quantify the amount of information contained in data. This places a limit on our ability to compress data. For a distribution P its *entropy*, $H[P]$, is defined as:

$$H[P] = \sum_j -P(j) \log P(j).$$

:eqlabel:`eq_softmax_reg_entropy`

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution P, we need at least $H[P]$ "nats" to encode it :cite:`Shannon.1948`. If you wonder what a "nat" is, it is the equivalent of bit but when using a code with base e rather than one with base 2. Thus, one nat is $\frac{1}{\log(2)} \approx 1.44$ bit.

Surprisal

You might be wondering what compression has to do with prediction. Imagine that we have a stream of data that we want to compress. If it is always easy for us to predict the next token, then this data is easy to compress. Take the extreme example where every token in the stream always takes the same value. That is a very boring data stream! And not only it is boring, but it is also easy to predict. Because the tokens are always the same, we do not have to transmit any information to communicate the contents of the stream. Easy to predict, easy to compress.

However if we cannot perfectly predict every event, then we might sometimes be surprised. Our surprise is greater when an event is assigned lower probability. Claude Shannon settled on $\log \frac{1}{P(j)} = -\log P(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $P(j)$. The entropy defined in :eqref:`eq_softmax_reg_entropy` is then the *expected surprisal* when one assigned the correct probabilities that truly match the data-generating process.

Cross-Entropy Revisited

So if entropy is the level of surprise experienced by someone who knows the true probability, then you might be wondering, what is cross-entropy? The cross-entropy *from* P *to* Q, denoted $H(P, Q)$, is the expected surprisal of an observer with subjective probabilities Q upon seeing data that was actually generated according to probabilities P. This is given by $H(P, Q) \stackrel{\text{def}}{=} \sum_j -P(j) \log Q(j)$. The lowest possible cross-entropy is achieved when $P = Q$. In this case, the cross-entropy from P to Q is $H(P, P) = H(P)$.

In short, we can think of the cross-entropy classification objective in two ways: (i) as maximizing the likelihood of the observed data; and (ii) as minimizing our surprisal (and thus the number of bits) required to communicate the labels.

Summary and Discussion

In this section, we encountered the first nontrivial loss function, allowing us to optimize over *discrete* output spaces. Key in its design was that we took a probabilistic approach, treating discrete categories as instances of draws from a probability distribution. As a side effect, we encountered the softmax, a convenient activation function that transforms outputs of an ordinary neural network layer into valid discrete probability distributions. We saw that the derivative of the cross-entropy loss when combined with softmax behaves very similarly to the derivative of squared error; namely by taking the difference between the expected behavior and its prediction. And, while we were only able to scratch the very surface of it, we encountered exciting connections to statistical physics and information theory.

While this is enough to get you on your way, and hopefully enough to whet your appetite, we hardly dived deep here. Among other things, we skipped over computational considerations. Specifically, for any fully connected layer with d inputs and q outputs, the parametrization and computational cost is $\mathcal{O}(dq)$, which can be prohibitively high in practice. Fortunately, this cost of transforming d inputs into q outputs can be reduced through approximation and compression. For instance Deep Fried Convnets :cite:`Yang.Moczulski.Denil.ea.2015` uses a combination of permutations, Fourier transforms, and scaling to reduce the cost from quadratic to log-linear. Similar techniques work for more advanced structural matrix approximations :cite:`sindhwani2015structured`. Lastly, we can use quaternion-like decompositions to reduce the cost to $\mathcal{O}(\frac{dq}{n})$, again if we are willing to trade off a small amount of accuracy for computational and storage cost :cite:`Zhang.Tay.Zhang.ea.2021` based on a compression factor n. This is an active area of research. What makes it challenging is that we do not necessarily strive for the most compact

representation or the smallest number of floating point operations but rather for the solution that can be executed most efficiently on modern GPUs.

Exercises

1. We can explore the connection between exponential families and softmax in some more depth.
    1. Compute the second derivative of the cross-entropy loss $l(\mathbf{y}, \hat{\mathbf{y}})$ for softmax.
    2. Compute the variance of the distribution given by $\mathrm{softmax}(\mathbf{o})$ and show that it matches the second derivative computed above.
2. Assume that we have three classes which occur with equal probability, i.e., the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    1. What is the problem if we try to design a binary code for it?
    2. Can you design a better code? Hint: what happens if we try to encode two independent observations? What if we encode n observations jointly?
3. When encoding signals transmitted over a physical wire, engineers do not always use binary codes. For instance, PAM-3 uses three signal levels $\{-1, 0, 1\}$ as opposed to two levels $\{0, 1\}$. How many ternary units do you need to transmit an integer in the range $\{0, \ldots, 7\}$? Why might this be a better idea in terms of electronics?
4. The Bradley--Terry model uses a logistic model to capture preferences. For a user to choose between apples and oranges one assumes scores o_{apple} and o_{orange}. Our requirements are that larger scores should lead to a higher likelihood in choosing the associated item and that the item with the largest score is the most likely one to be chosen :cite:`Bradley.Terry.1952`.
    1. Prove that softmax satisfies this requirement.
    2. What happens if you want to allow for a default option of choosing neither apples nor oranges? Hint: now the user has three choices.
5. Softmax gets its name from the following mapping: $\mathrm{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
    1. Prove that $\mathrm{RealSoftMax}(a, b) > \max(a, b)$.
    2. How small can you make the difference between both functions? Hint: without loss of generality you can set $b = 0$ and $a \geq b$.
    3. Prove that this holds for $\lambda^{-1}\mathrm{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
    4. Show that for $\lambda \to \infty$ we have $\lambda^{-1}\mathrm{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
    5. Construct an analogous softmin function.
    6. Extend this to more than two numbers.
6. The function $g(\mathbf{x}) \stackrel{\mathrm{def}}{=} \log \sum_i \exp x_i$ is sometimes also referred to as the log-partition function.
    1. Prove that the function is convex. Hint: to do so, use the fact that the first derivative amounts to the probabilities from the softmax function and show that the second derivative is the variance.
    2. Show that g is translation invariant, i.e., $g(\mathbf{x} + b) = g(\mathbf{x})$.
    3. What happens if some of the coordinates x_i are very large? What happens if they're all very small?
    4. Show that if we choose $b = \max_i x_i$ we end up with a numerically stable implementation.
7. Assume that we have some probability distribution P. Suppose we pick another distribution Q with $Q(i) \propto P(i)^\alpha$ for $\alpha > 0$.
    1. Which choice of α corresponds to doubling the temperature? Which choice corresponds to halving it?
    2. What happens if we let the temperature approach 0?
    3. What happens if we let the temperature approach ∞?

Discussions

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

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()
```

Loading the Dataset

```
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))
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-u
100%|██████████| 26421880/26421880 [00:02<00:00, 9489696.50it/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-u
100%|██████████| 29515/29515 [00:00<00:00, 137218.41it/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-ub
100%|██████████| 4422102/4422102 [00:01<00:00, 2527271.45it/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-ub
100%|██████████| 5148/5148 [00:00<00:00, 19736999.08it/s]Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/r


(60000, 10000)
```

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

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

```
@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]
```

Reading a Minibatch

```
@d2l.add_to_class(FashionMNIST)
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)
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:560: UserWarning: This DataLoader will create 4 worker processes in total
  warnings.warn(_create_warning_msg(
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

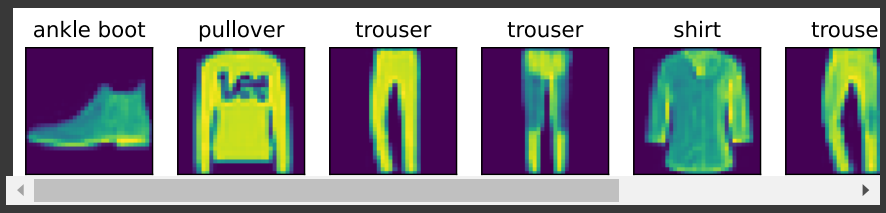
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'10.21 sec'
```

Visualization

```
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)
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)
```



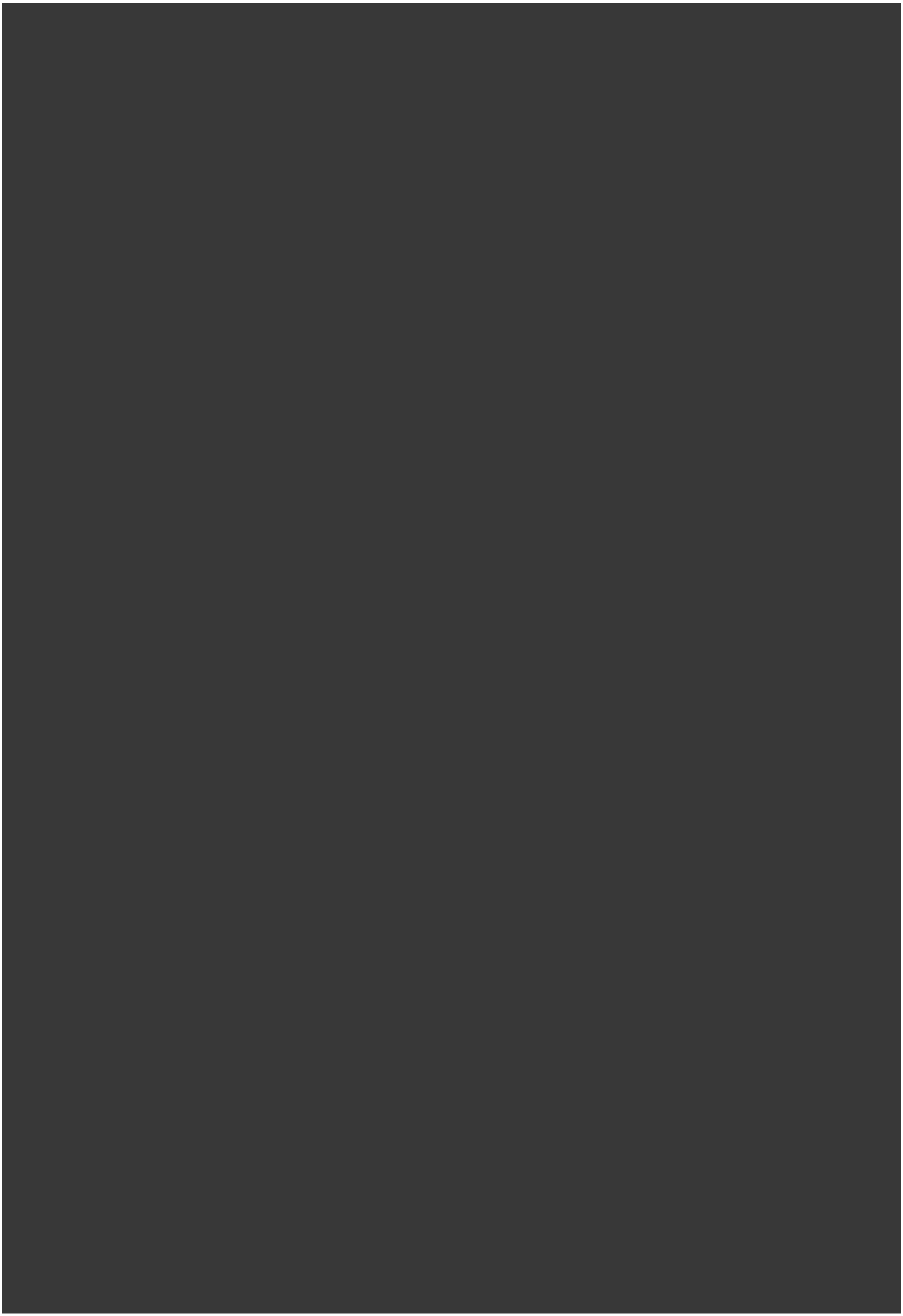
Summary

We now have a slightly more realistic dataset to use for classification. Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories. We will use this dataset in subsequent sections and chapters to evaluate various network designs, from a simple linear model to advanced residual networks. As we commonly do with images, we read them as a tensor of shape (batch size, number of channels, height, width). For now, we only have one channel as the images are grayscale (the visualization above uses a false color palette for improved visibility).

Lastly, data iterators are a key component for efficient performance. For instance, we might use GPUs for efficient image decompression, video transcoding, or other preprocessing. Whenever possible, you should rely on well-implemented data iterators that exploit high-performance computing to avoid slowing down your training loop.

Discussion

```
'''
- 대표적인 이미지 분류 태스크 데이터셋으로 MNIST 계열 데이터셋들이 있다.]
- 전통적으로 이미지는 c×h×w 텐서로 저장된다. (c  is the number of color channels,  h  is the height and  w  is the width.)
'''
```



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

The Base Classification Model

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

The `Classifier` Class

```
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)
```

Accuracy

```
@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
```

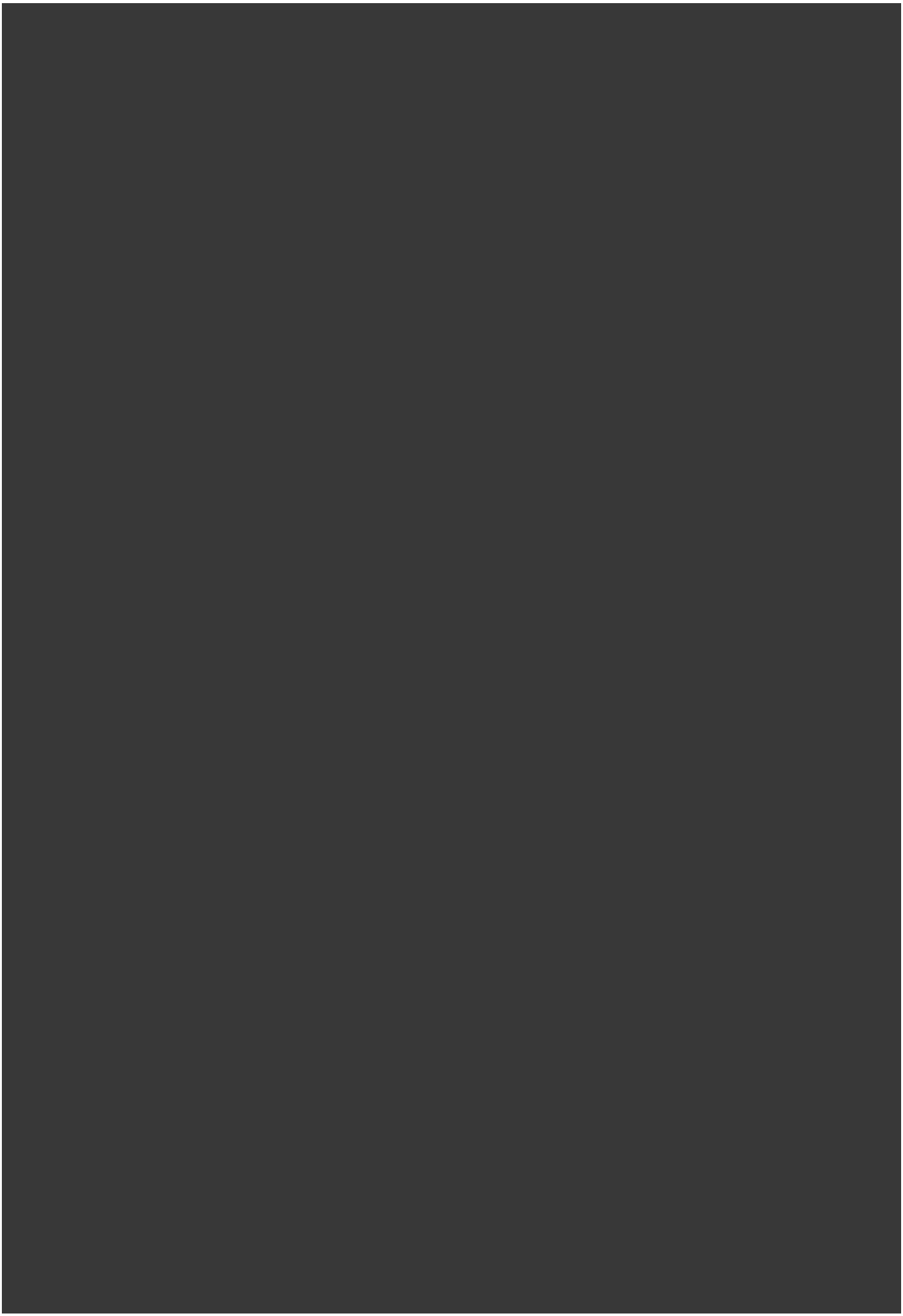
```
## Discussion (TakeAway Msg)
```

Discussion (TakeAway Msg)

```
'''
- 예측 클래스를 결정할때, 각 클래스에 대한 일종의 예측점수인 entropy 값 (prediction score) 들 중에서 argmax 메소드를 통해 가장 높은 확률 값을 지닌 클라
- 각 validatino step 마다 loss value 와 cls accuracy 를 산출한다.
'''
```

Summary

Classification is a sufficiently common problem that it warrants its own convenience functions. Of central importance in classification is the *accuracy* of the classifier. Note that while we often care primarily about accuracy, we train classifiers to optimize a variety of other objectives for statistical and computational reasons. However, regardless of which loss function was minimized during training, it is useful to have a convenience method for assessing the accuracy of our classifier empirically.



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

Softmax Regression Implementation from Scratch

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

The Softmax

```
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([[0.1580, 0.1797, 0.1709, 0.2413, 0.2502],
         [0.1736, 0.3434, 0.1762, 0.1536, 0.1532]]),
 tensor([1.0000, 1.0000]))
```

The Model

```
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)
```

The Cross-Entropy Loss

```
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])
```

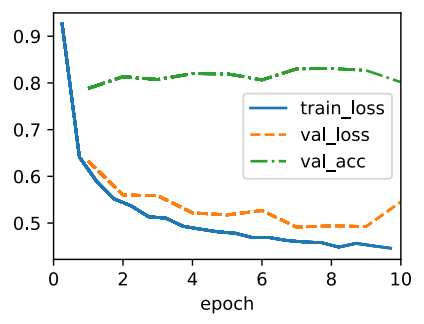
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

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)
```

Training

```
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)
```

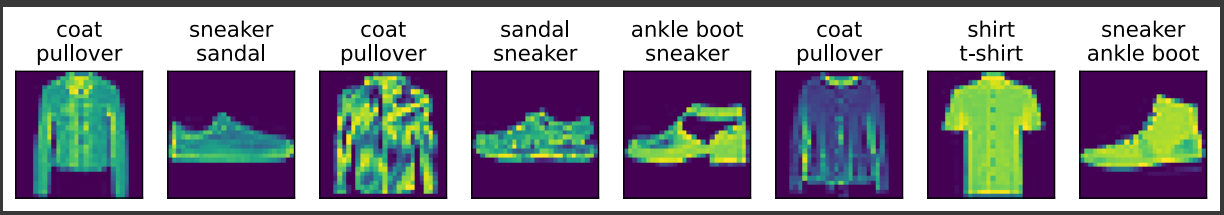


Prediction

```
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)
```

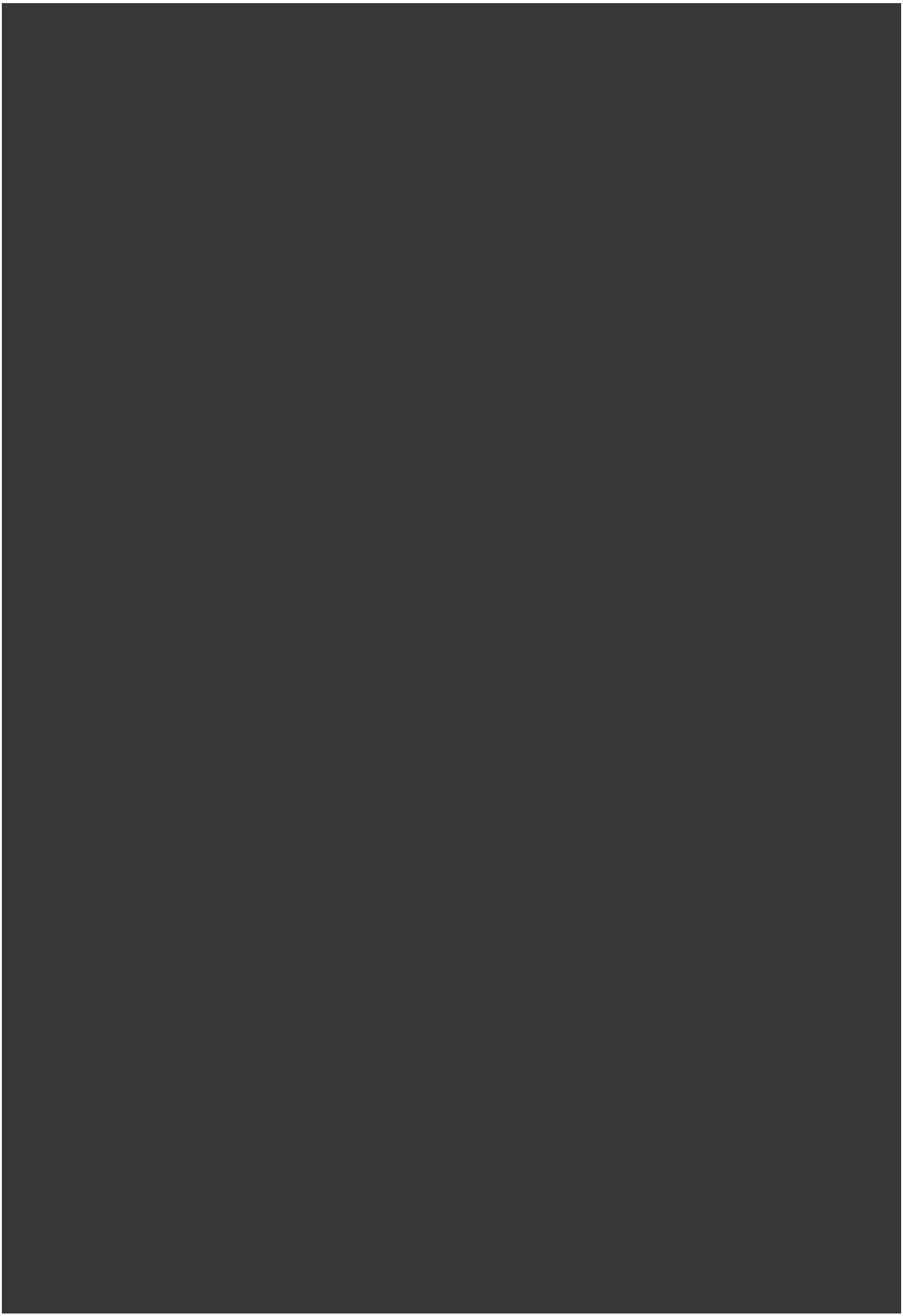


Discussion (TakeAway Msg)

```
'''
- 소프트맥스 회귀에서는 네트워크의 # of ouputs 과 # of classes 가 같아야 한다.
- 크로스엔트로피는 log-likelihodd 음수값도 취한다.
'''
```

Summary

By now we are starting to get some experience with solving linear regression and classification problems. With it, we have reached what would arguably be the state of the art of 1960--1970s of statistical modeling. In the next section, we will show you how to leverage deep learning frameworks to implement this model much more efficiently.



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

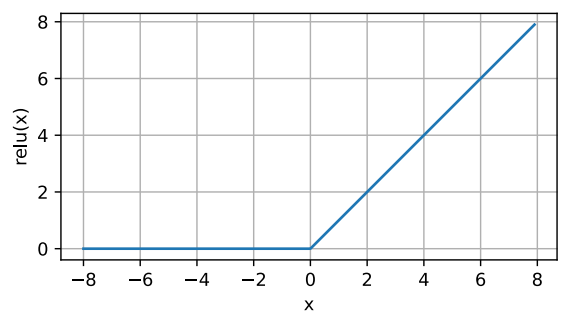
Multilayer Perceptrons

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

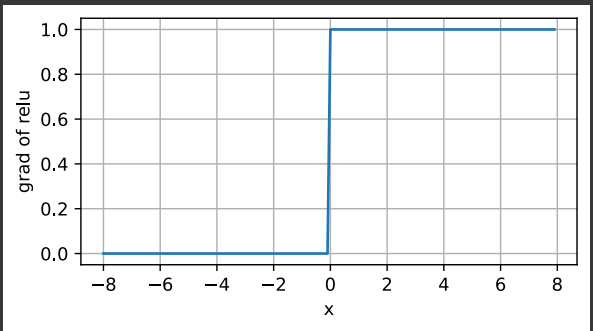
Activation Functions

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))
```

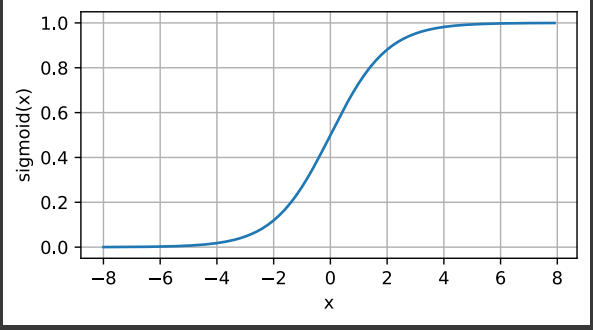



```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```

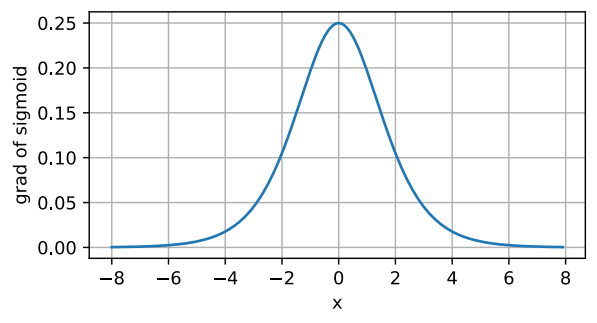


Sigmoid

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

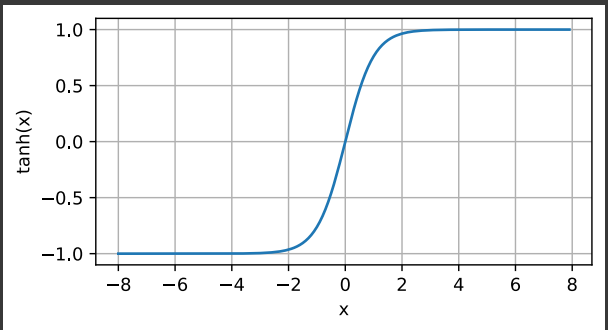



```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

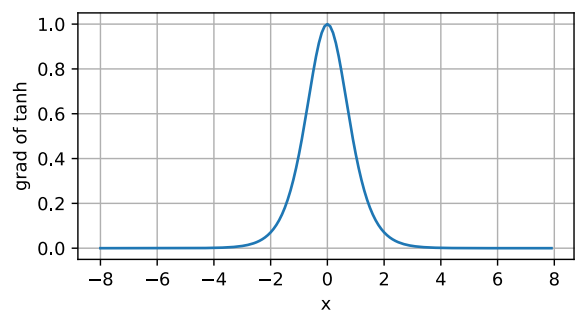


Tanh Function

```
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))
```



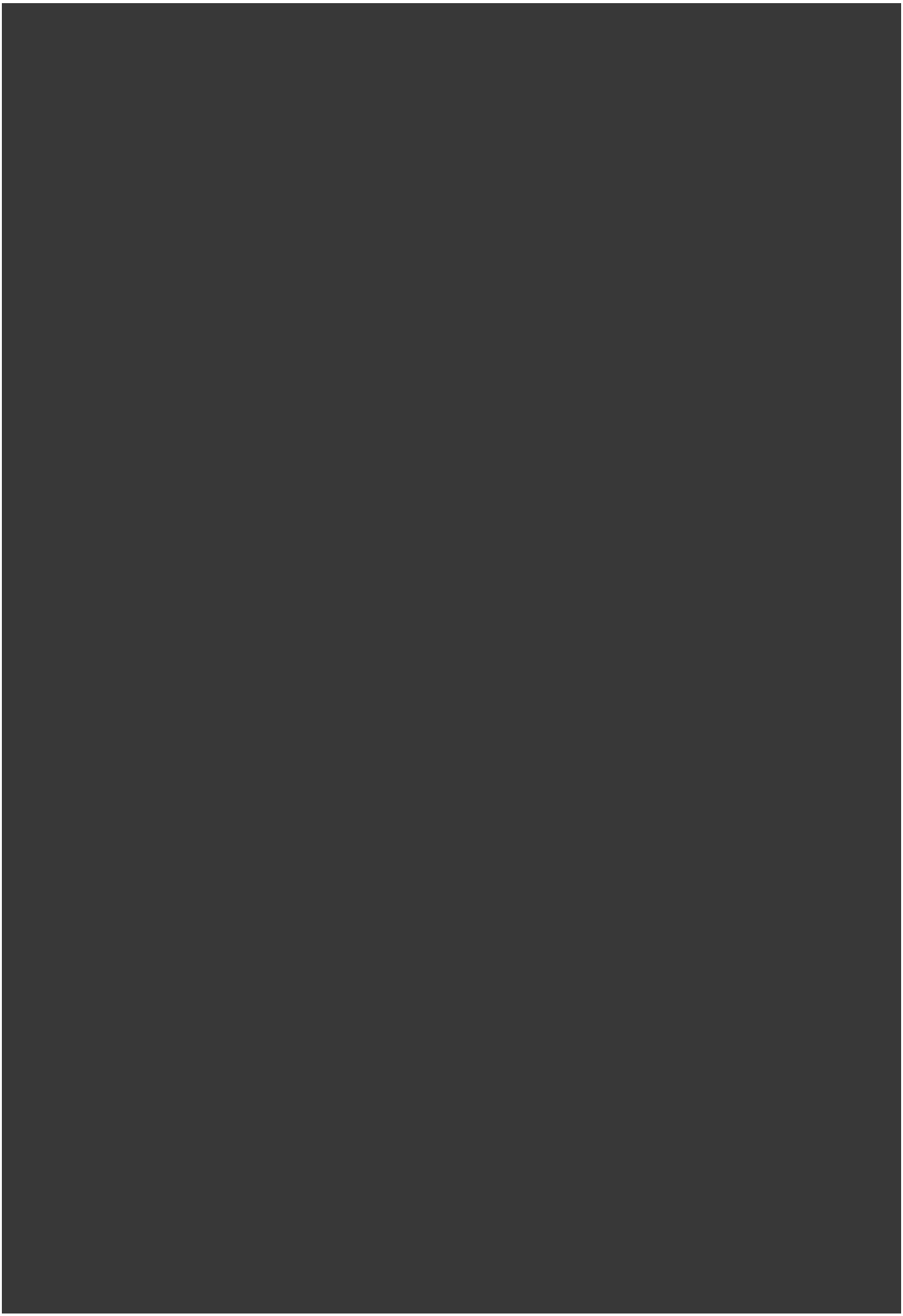
Summary and Discussion

We now know how to incorporate nonlinearities to build expressive multilayer neural network architectures. As a side note, your knowledge already puts you in command of a similar toolkit to a practitioner circa 1990. In some ways, you have an advantage over anyone working back then, because you can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code. Previously, training these networks required researchers to code up layers and derivatives explicitly in C, Fortran, or even Lisp (in the case of LeNet).

A secondary benefit is that ReLU is significantly more amenable to optimization than the sigmoid or the tanh function. One could argue that this was one of the key innovations that helped the resurgence of deep learning over the past decade. Note, though, that research in activation functions has not stopped. For instance, the GELU (Gaussian error linear unit) activation function $x\Phi(x)$ by :citet:`Hendrycks.Gimpel.2016` ($\Phi(x)$ is the standard Gaussian cumulative distribution function) and the Swish activation function $\sigma(x) = x\,\mathrm{sigmoid}(\beta x)$ as proposed in :citet:`Ramachandran.Zoph.Le.2017` can yield better accuracy in many cases.

Discussion (TakeAway Msg)

```
'''
각 활성화함수마다의 특징이 다르고, 각 태스크 종류마다 적합한 활성화함수의 종류가 다르다.
다양한 실험 및 리서치 조사를 통해 그때 그때 필요에 맞는 적절한 활성화 함수를 취할 필요가 있어 보인다.
'''
```



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

Implementation of Multilayer Perceptrons

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

Initializing Model Parameters

```
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))
```

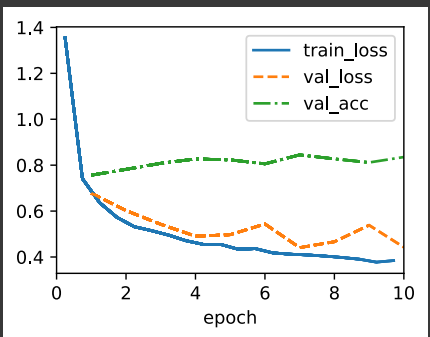
Model

```
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
```

Training

```
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)
```



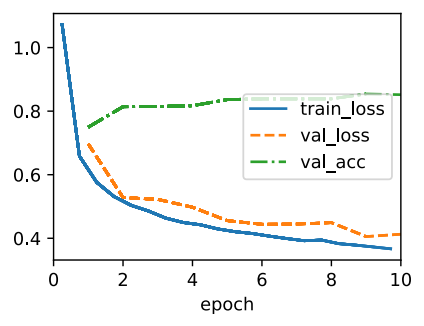
Concise Implementation

Model

```
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))
```

Training

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



Discussion (TakeAway Msg)

```
'''
- MLP 는 hidden layer 의 개수 등 특히 하이퍼파라미터 튜닝을 할 여지가 많은 것 같다.
- MLP 모델은 앞선 회귀 모델들과 달리 '2개'의 전결합층이 네트워크에 더해져 있다. 첫번째는 hidden layer 이고, 두번째는 output layer 이다.
'''
```

Summary

Now that we have more practice in designing deep networks, the step from a single to multiple layers of deep networks does not pose such a significant challenge any longer. In particular, we can reuse the training algorithm and data loader. Note, though, that implementing MLPs from scratch is nonetheless messy: naming and keeping track of the model parameters makes it difficult to extend models. For instance, imagine wanting to insert another layer between layers 42 and 43. This might now be layer 42b, unless we are willing to perform sequential renaming. Moreover, if we implement the network from scratch, it is much more difficult for the framework to perform meaningful performance optimizations.

Nonetheless, you have now reached the state of the art of the late 1980s when fully connected deep networks were the method of choice for neural network modeling. Our next conceptual step will be to consider images. Before we do so, we need to review a number of statistical basics and details on how to compute models efficiently.

Discussion (TakeAway Msg)

```
'''
역전파 계산 과정 한글 참고자료: https://wikidocs.net/37406
코드 상 뿐만 아니라, 실제 수식 상으로도 직접 역전파 과정을 계산하기 위해 연습이 필요할 것 같다.
'''
```

Forward Propagation, Backward Propagation, and Computational Graphs

:label: `sec_backprop`

So far, we have trained our models with minibatch stochastic gradient descent. However, when we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. When it came time to calculate the gradients, we just invoked the backpropagation function provided by the deep learning framework.

The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models required recalculating complicated derivatives by hand. Surprisingly often, academic papers had to allocate numerous pages to deriving update rules. While we must continue to rely on automatic differentiation so we can focus on the interesting parts, you ought to know how these gradients are calculated under the hood if you want to go beyond a shallow understanding of deep learning.

In this section, we take a deep dive into the details of *backward propagation* (more commonly called *backpropagation*). To convey some insight for both the techniques and their implementations, we rely on some basic mathematics and computational graphs. To start, we focus our exposition on a one-hidden-layer MLP with weight decay (ℓ_2 regularization, to be described in subsequent chapters).

Forward Propagation

Forward propagation (or *forward pass*) 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. We now work step-by-step through the mechanics of a neural network with one hidden layer. This may seem tedious but in the eternal words of funk virtuoso James Brown, you must "pay the cost to be the boss".

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and that our hidden layer does not include a bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After running the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ through the activation function ϕ we obtain our hidden activation vector of length h:

$$\mathbf{h} = \phi(\mathbf{z}).$$

The hidden layer output $\mathbf{h}$ is also an intermediate variable. Assuming that the parameters of the output layer possess only a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector of length q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

Assuming that the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y).$$

As we will see the definition of ℓ_2 regularization to be introduced later, given the hyperparameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_{\mathrm{F}}^2 + \|\mathbf{W}^{(2)}\|_{\mathrm{F}}^2\right),$$

:eqlabel: `eq_forward-s`

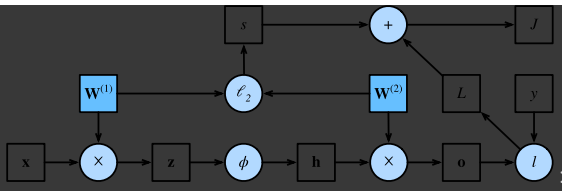
where the Frobenius norm of the matrix is simply the ℓ_2 norm applied after flattening the matrix into a vector. Finally, the model's regularized loss on a given data example is:

$$J = L + s.$$

We refer to J as the *objective function* in the following discussion.

Computational Graph of Forward Propagation

Plotting *computational graphs* helps us visualize the dependencies of operators and variables within the calculation. :numref: `fig_forward` contains the graph associated with the simple network described above, where squares denote variables and circles denote operators. The lower-left corner signifies the input and the upper-right corner is the output. Notice that the directions of the arrows (which illustrate data flow) are primarily rightward and upward.



:label: `fig_forward`

Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In short, the method traverses the network in reverse order, from the output to the input layer, according to the *chain rule* from calculus. The algorithm stores any intermediate variables (partial derivatives) required while calculating the gradient with respect to some parameters. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z with respect to X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right).$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions, have been carried out. For vectors, this is straightforward: it is simply matrix–matrix multiplication. For higher dimensional tensors, we use the appropriate counterpart. The operator prod hides all the notational overhead.

Recall that the parameters of the simple network with one hidden layer, whose computational graph is in :numref: `fig_forward`, are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accomplish this, we apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the computational graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s:

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1.$$

Next, we compute the gradient of the objective function with respect to variable of the output layer $\mathbf{o}$ according to the chain rule:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q.$$

Next, we calculate the gradients of the regularization term with respect to both parameters:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

Now we are able to calculate the gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}.$$

:eqlabel: `eq_backprop-J-h`

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer output $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}.$$

Since the activation function ϕ applies elementwise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}$ requires that we use the elementwise multiplication operator, which we denote by $\odot$:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right).$$

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

Training Neural Networks

When training neural networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the computational graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed.

Take the aforementioned simple network as an illustrative example. On the one hand, computing the regularization term :eqref: `eq_forward-s` during forward propagation depends on the current values of model parameters $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. They are given by the optimization algorithm according to backpropagation in the most recent iteration. On the other hand, the gradient calculation for the parameter :eqref: `eq_backprop-J-h` during backpropagation depends on the current value of the hidden layer output $\mathbf{h}$, which is given by forward propagation.

Therefore when training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. Note that backpropagation reuses the stored intermediate values from forward propagation to avoid duplicate calculations. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why training requires significantly more memory than plain prediction. Besides, the size of such intermediate values is roughly proportional to the number of network layers and the batch size. Thus, training deeper networks using larger batch sizes more easily leads to *out-of-memory* errors.

Summary

Forward propagation sequentially calculates and stores intermediate variables within the computational graph defined by the neural network. It proceeds from the input to the output layer. Backpropagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order. When training deep learning models, forward propagation and backpropagation are interdependent, and training requires significantly more memory than prediction.