

✓ COSE474-2024F : Deep Learning

✓ Installation

더블클릭 또는 Enter 키를 눌러 수정

```
pip install torch==2.0.0 torchvision==0.15.1
```

 숨겨진 출력 표시

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


 숨겨진 출력 표시

✓ 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([[0.3160, -0.6630, 0.4055, 0.4006],
 [-0.8909, 0.9642, -3.4818, -0.6631],
 [-1.3800, -0.3388, 1.5111, -0.4506]])

```
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.])))
```

여기서 X는 위의 `X = x.reshape(3,4)`의 결과 `tensor([[0., 1., 2., 3.], [4., 5., 6., 7.], [8., 9., 10., 11.]])`를 뜻함. 해당 결과에서의 행을 뽑아내는 것
임 행렬의 특정 값을 변경한 후 출력하는 것이 가능하다는 것을 파악함

이 챕터로 모든 행렬을 구현하는 것이 가능하다는 것을 이해함.

```
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.])))
```

행렬을 기존 행렬에 붙이는 방법임.

```
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): 138526734076240
   id(Z): 138526734076240
```

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

```
↔ True
```

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

```
↔ (numpy.ndarray, torch.Tensor)
```

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

```
↔ (tensor([3.5000]), 3.5, 3.5, 3)
```

tensor내의 element들을 다루는데 초점을 둔 chapter였음.

2.1 Discussion

tensor내의 숫자를 수정하는 것을 자유자재로 다룰 수 있음

tensor내의 element들의 exp, 사칙 연산을 시도해 보았음. tensor내의 element들의 값이 같은지 파악하는 것 등을 통해 이들로 구현해볼 수 있는 여러 작업들이 생각났음.

tensor의 행렬을 조정하고, 행렬 연산을 하게 만드는 과정을 시도해봤음

tensor 내의 변수의 자료형 조정도 가능하다는 점을 알게 되었음

Tensor를 이용해서 elements를 포함하는 것을 봄 size를 파악하고, 이를 행렬로 만드는 것을 보고, 행렬이 컴퓨터과학과 연결되는 점을 볼 수 있었음.

2.2 Data Preprocessing

```
import os
```

```
os.makedirs(os.path.join('..', 'data'), exist_ok=True)
```

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

```
import pandas as pd
```

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

```
↕
```

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

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

```
↕
```

	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

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

```
↕
```

	NumRooms	RoofType_Slate	RoofType_nan
0	3.0	False	True
1	2.0	False	True
2	4.0	True	False
3	3.0	False	True

```
import torch
```

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

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

전체적으로 앞으로 훈련에 사용될 dataframe을 다루는 기초적인 방법을 소개하고 있음.

2.2 Discussion

data set을 읽는 예제임. 이는 지도 학습에 사용됨. 구체적으로는, input value와 상응되는 target value를 구분해내기 위해 존재함. 값이 이진인 형태를 이루면, 이를 True, False로 구분해 낼 수 있음.

missing value에 한해서는, 나머지 값들의 평균을 그대로 missing value에 넣어주는 코드가 존재함

모든 input, target 값들이 수들이면 이들을 tensor에 넣을 수 있음.

이들을 통해, 어떻게 data column들을 구분하고, missing variables들을 채우며, pandas data를 tensor에 삽입할 수 있게 되었음.

✓ 2.3 Linear Algebra

```
import torch
```

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

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

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

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

```
→ tensor([0, 1, 2])
```

```
x[2]
```

```
→ tensor(2)
```

이러한 코드는 리스트의 문법과 매우 비슷함

```
len(x)
```

```
→ 3
```

```
x.shape
```

```
→ torch.Size([3])
```

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

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

행렬의 **dimensionality**를 조정할 수 있음.

```
A.T
```

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

행렬의 행과 열을 바꿔 표현하는 것이 가능함

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

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

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

```
→ tensor([[[[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]],
            [[12, 13, 14, 15],
              [16, 17, 18, 19],
              [20, 21, 22, 23]]]])
```

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone() # Assign a copy of A to B by allocating new memory
A, A + B
```

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

```
A * B
```

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

행렬의 연산이 아니라, 같은 위치에 존재하는 각 행렬의 element들을 그대로 같은 차원에 연산하여 넣는 개념임

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

```
↔ (tensor([[[ 2,  3,  4,  5],
             [ 6,  7,  8,  9],
             [10, 11, 12, 13]],
           [[14, 15, 16, 17],
            [18, 19, 20, 21],
            [22, 23, 24, 25]]]),
    torch.Size([2, 3, 4]))
```

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

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

```
A.shape, A.sum()
```

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

A는 (tensor([[0., 1., 2.], [3., 4., 5.]])

임. shape, sum을 파악해보면 각각 들어 맞다는 것을 알 수 있음

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

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

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

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

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

```
↔ tensor(True)
```

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

```
↔ (tensor(2.5000), tensor(2.5000))
```

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

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

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

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

```
A / sum_A
```

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

한 tensor의 행렬에서 sum, mean, shape등을 출력할 수 있음

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

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

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

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

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

```
↔ tensor(3.)
```

(tensor([0., 1., 2.]), tensor([1., 1., 1.]) 각 element를 곱하고, 이를 더한 것이 3이라는 것을 알 수 있음.

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

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

mv, @가 행렬의 사칙연산 곱을 나타냈다는 걸 알 수 있음.

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

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

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

```
↔ tensor(5.)
```

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

```
↔ tensor(7.)
```

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

```
↔ tensor(6.)
```

2.3 Discussion

tensor의 scalar들에 대해 사칙 연산이 가능함. 이 scalar들이 모여 vector를 구성하게 됨. vector들이 dataset들을 의미할 때, 각각의 값들은 dataset의 값들임. 이들은 **zero-based indexing**으로 기본적으로 0부터 시작됨.

이 indexing으로 우리는 vector로부터 값을 추출할 수 있음. 추가로, len을 활용하여 몇개의 원소를 포함하고 있는지를 알고서, 이 vector의 dimensionality를 파악할 수 있음. 이 len값은 shape를 통해서도 파악 가능함.

이 vector 값을 matrix화 할 수 있음. 기존에 len을 출력했다면, 이를 2개의 값의 곱으로 reshape 할 수 있음. 이와 같은 코드도 구현했음. 이 구현과정에서 행렬의 축을 바꿀 수 도 있음.

행렬을 구성했다면, 수학에서 간단히 배운대로, 행렬의 연산을 수행하는 것도 가능함. 또한, 행렬의 원소를 삭제하거나, 이들의 합, 평균, 개수를 연산하는 것도 가능하게 됨. 물론, 행렬을 유지한 채로도 여러 연산들을 수행할 수 있음.

지금까지의 구현으로 선형 대수를 colab을 통해 구현할 수 있음. 딥러닝 수업 때, 학습했던, weight와 변수들간의 곱들의 합을 구한다거나 할 수 있음

이와 같은 선형대수 구현은 앞으로 머신 러닝을 할 때, dataset의 구조를 발견하거나, 예측 문제에 기여할 수 있음.

✓ 2.5 Automatic Differentiation

```
import torch
```

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

```
↔ tensor([0., 1., 2., 3.])
```

```

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

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

⇒ tensor(28., grad_fn=<MulBackward0>)

y.backward()
x.grad

⇒ tensor([ 0.,  4.,  8., 12.])

x.grad == 4 * x

⇒ tensor([True, True, True, True])

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

⇒ tensor([1., 1., 1., 1.])

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

⇒ tensor([0., 2., 4., 6.])

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

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

⇒ tensor([True, True, True, True])

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

⇒ tensor([True, True, True, True])

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

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

a.grad == d / a

⇒ tensor(True)

```

2.5 Discussion

여기서는 `gradient`를 출력하는 방법을 먼저 배움. 먼저 함수를 계산하고, 이를 역으로 추출하는 것이 가능함. 이들의 톨로, 자동적으로 혹은 수동으로 미분값을 계산할 수 있음. 추가로, 이러한 방법으로 `model`을 최적화할 수도 있음.

이들을 통해, 1) 미분을 원하는 변수에 `gradient`를 추가하고, 2) 목표값을 계산해 저장한 후 3) `backpropagation` 함수를 실행하여, 4) `gradient`를 찾을 수 있는 루틴을 구성함.

해당 chapter는 이후에 나올 `backpropagation`에 대비한 내용임. `gradient`를 찾는 루틴에 대한 내용이 코드로 구현되어 있음.

3.1 Linear Regression

```
pip install d2l
```


 숨겨진 출력 표시

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

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

10000개의 1이 담긴 `tensor`를 두 개 생성함.

a, b

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

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

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

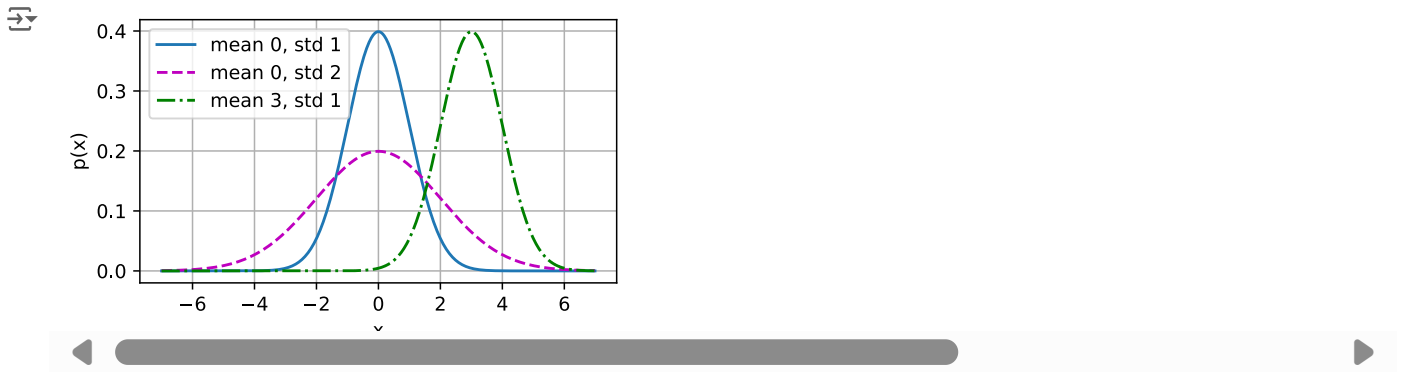
 

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

정규화 함수를 직접 구현함

```
# 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])
```



정규화 함수를 그래픽에 그려봄

3.1 Discussion

선형 회귀에도 활용할 수 있음. Training set을 바탕으로 label을 추정하는 것을 할 수 있다는 것임. 이 때 Training set의 값들의 바탕이 되는 variable들을 feature라고도 부름.

모든 feature들을 vector에 모으고, 이들의 가중치를 벡터에 모아서, 곱을 통해 모델을 간결하게 표현할 수 있음.

Loss function을 통해 데이터를 적합하도록 만드는 척도를 세울 수 있음. 다시 말해 손실 함수를 통해 실제 값과 예측된 목표값 사이의 차이를 정량화 할 수 있음. 0에 가까울 수록 좋은 지표임.

이러한 선형 회귀는 단일 계층의 신경망으로 치환할 수 있음. 여기서 출발하여 더 많은 계층을 가진 네트워크를 접하게 될 것임. 이들을 결합시키면 더욱 복잡한 신경망을 계산할 수 있게 됨.

3.2 Object-Oriented Design for Implementation

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

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

setattr는 파이썬에서 객체의 속성을 설정하는 데 사용되는 내장 함수임.

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

```
a = A()
```

init은 파이썬에서 클래스의 생성자 메서드임. 클래스의 인스턴스가 생성될 때 자동으로 호출되며, 주로 인스턴스 변수를 초기화하는 데 사용됨.

```
@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
```

hasattr는 파이썬의 내장 함수로, 특정 객체가 주어진 속성을 가지고 있는지 확인하는 데 사용됨.

```
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 NotImplementedError
```

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



```
class Module(nn.Module, d2l.HyperParameters):
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, 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 initied'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / W
            self.trainer.num_train_batches
            n = self.trainer.num_train_batches / W
            self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / W
            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

```

plot_train_per_epoch는 훈련 중 몇 번의 에포크마다 훈련 결과를 시각화할지를 나타내고, 기본값을 2로 설정한 것임. plot_valid_per_epoch는 검증 중 몇 번의 에포크마다 검증 결과를 시각화할지를 나타내고, 기본값은 1임.

self.board = ProgressBoard():

ProgressBoard는 훈련 과정의 진행 상황을 추적하거나 시각화하는 데 사용되는 객체임. 이 객체는 모델 훈련 중에 유용한 정보를 기록하고 시각화하는 데 도움을 줌. self.board는 이 객체를 인스턴스 변수로 저장하여, 클래스의 다른 메서드에서도 사용할 수 있게 함

시각화할 그래프의 x축 레이블을 'epoch'로 설정함. train 상태인지, 검증 상태인지에 따라 다른 방식으로 x축 값을 계산함.

훈련 모드 (train이 True인 경우):

x = self.trainer.train_batch_idx / self.trainer.num_train_batches: 현재 배치 인덱스를 전체 훈련 배치 수로 나누어, 0과 1 사이의 값으로 정규화함. 이는 현재 훈련의 진행 상황을 나타냅니다. n = self.trainer.num_train_batches / self.plot_train_per_epoch: 전체 훈련 배치 수를 훈련 시각화 간격으로 나누어, 몇 배치마다 점을 그릴지를 계산함. 검증 모드 (train이 False인 경우):

x = self.trainer.epoch + 1: 현재 에포크 수에 1을 더해 x축 값을 설정함. n = self.trainer.num_val_batches / self.plot_valid_per_epoch: 전체 검증 배치 수를 검증 시각화 간격으로 나누어 몇 배치마다 점을 그릴지를 계산함.

self.board.draw(...):

self.board.draw 메서드를 호출하여 그래프에 점을 추가함. 인자로는 x축 값(x), y축 값(value.to(d2l.cpu()).detach().numpy()), 레이블, 그리고 점을 그릴 간격(every_n=int(n))이 포함됨. value.to(d2l.cpu()).detach().numpy()는 텐서를 CPU로 옮기고, 그래디언트 추적을 끊은 후 NumPy 배열로 변환하여 시각화에 사용하게 됨. 결론적으로, 이 메서드는 주어진 지표(value)를 훈련 또는 검증 과정에서 시각화하기 위해 적절한 x축 값과 간격으로 그래프에 점을 추가함.

Batch (배치) 정의: 데이터셋을 여러 개의 작은 그룹으로 나눈 것을 의미함. 예를 들어, 전체 데이터셋이 1,000개 샘플이라면, 이를 100개 샘플씩 10개의 배치로 나눌 수 있음. 목적: 한 번에 모든 데이터를 처리하는 것이 아니라, 메모리 효율성을 높이고 학습 속도를 증가시키기 위해 사용됨. 각 배치마다 모델이 업데이트되며, 이 과정을 통해 전체 데이터셋의 경향을 반영함. Epoch (에포크) 정의: 전체 데이터셋이 모델을 한 번 완전히 통과한 횟수를 의미함. 예를 들어, 1 에포크는 모델이 모든 학습 샘플을 한 번 사용하여 가중치를 업데이트한 것임. 목적: 에포크 수는 모델이 데이터를 얼마나 많이 학습했는지를 나타내며, 보통 여러 에포크 동안 훈련하여 모델의 성능을 향상시킴.

이렇게 배치와 에포크를 활용하여 모델을 효과적으로 향상시킬 수 있음.

```

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)

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

```

이 클래스는 딥러닝 모델을 학습하기 위한 기본 클래스를 정의하고 있음.

max_epochs는 훈련할 최대 에포크 수, num_gpus는 사용할 GPU 수, gradient_clip_val은 그래디언트 클리핑 값을 의미함.

assert는 파이썬의 내장 키워드로, 조건식이 True일 때는 아무런 동작을 하지 않고, False일 경우 AssertionError를 발생시킴. 주로 디버깅이나 조건 검사를 위해 사용됨. 다시 말해, GPU 지원이 없음을 확인하고, 만약 GPU가 지정되면 오류를 발생시킴.

prepare_data(self, data)는 데이터 로더를 초기화 하고, 각각의 배치 수를 계산함.

model.trainer = self는 모델에 현 trainer를 설정하고, 나중에 참조할 수 있도록 함. model.board.xlim = [0, self.max_epochs]는 그래프의 x축 범위를 설정함. self.model = model로 trainer의 인스턴스 변수로 모델을 저장함.

마지막으로 fit(self, model, data)에서 model은 훈련할 모델, data는 데이터셋임. self.prepare_data(data): 데이터를 준비.

self.prepare_model(model): 모델을 준비. self.optim = model.configure_optimizers(): 모델의 최적화 기법을 설정함. 여러 에포크 동안 훈련을 수행하는 루프를 실행함 (self.fit_epoch()를 호출).

이 Trainer 클래스는 모델 훈련을 위한 구조를 제공함. 데이터와 모델을 초기화하고, 에포크 루프를 통해 훈련을 관리함.

3.2 Discussion

딥러닝의 구성 요소를 객체화 하여, 이들의 상호작용에 대해 클래스를 정의하는 과정을 거치는 것임. 이 장에서는 세 개의 클래스 1) 손실, 최적화 방법을 포함하는 모델 2) 훈련 및 검증을 위한 데이터모듈 3) 이 두가지를 Trainer 클래스를 통해 결합하기에 대해 다루게 됨.

클래스 구현 예시에서 이들이 실제 그래픽에서 sin과 cos을 그리는 것을 보았음. 이 툴의 실용성을 볼 수 있었음.

우리가 구현할 모든 모델의 기본 클래스인 Module클래스를 학습했음. 학습 가능한 매개 변수를 저장하는 **init**, 데이터 배치를 받아 손실 값을 반환하는 training_step, 학습 가능한 매개변수를 업데이트하는데 사용되는 최적화 방법을 반환하는 configure_optimizers를 작성해보았음. 모델의 전체적인 구성과 구조를 파악할 수 있었음.

다음으로 DataModule클래스도 작성해봤음 데이터를 준비하는 **init**메소드, 훈련 데이터셋을 위한 dataloader를 반환하는 train_dataloader를 보았고, 이들의 체계성을 파악할 수 있었음.

마지막으로, 데이터를 훈련 시키는 trainer클래스를 보았음.

이들을 통해 딥러닝 구현을위한 객체 지향 설계를 꼭 파악해볼 수 있었음. 이들이 데이터를 저장하고 서로 상호작용하는 모습을 그려볼 수 있었음. 이들이 d2l 라이브러리에 저장되어 있기에, 최적화기법, 모델, 데이터셋을 바꿔가며 보다 간단히 딥러닝을 구현하는게 가능하다는 점은 매우 편리해 보였음. 딥러닝의 개략적인 부분을 파악할 수 있는 챕터였음.

가장 적절한 batch와 epoch를 찾는 방법이 무엇일지에 대해 생각해보았음. 여러 검색을 통해 확인해본 결과 배치 크기는 일반적으로 32에서 256 사이에서 시도하며, 사용 가능한 메모리에 따라 선택하고, 에포크 수는 Early Stopping 기법을 사용하거나 학습 곡선을 참고하여 과적합을 방지할 수 있음.

✓ 3.4 Linear Regression Implementation from Scratch

```

%matplotlib inline
import torch
from d2l import torch as d2l

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)

```

num_inputs: 입력 데이터의 특성 수 **lr**: 학습률(learning rate) **sigma**: 초기 가중치를 설정하기 위한 표준 편차(기본값은 0.01)

self.b = torch.zeros(1, requires_grad = True): 절편(바이어스)인 **b**를 0으로 초기화함. **requires_grad=True**를 설정하여 이 변수가 gradient 계산에 포함되도록 함.

self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad = True): 이는 가중치 **w**를 초기화함. 이는 평균이 0이고 표준 편차가 **sigma**인 정규분포에서 가중치를 생성함.

이 클래스는 선형 회귀 모델의 기본 구성 요소인 가중치와 절편을 정의하며, 학습에 필요한 하이퍼파라미터를 저장함. 초기 가중치는 정규 분포를 사용하여 무작위로 설정되고, 바이어스는 0으로 초기화됨. 이렇게 초기화된 모델은 이후 훈련을 통해 가중치와 바이어스를 조정하여 데이터를 학습함.

```

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

```

matmul은 행렬 곱셈을 말함. **torch.matmul**은 PyTorch에서 두 개의 텐서를 곱할 때 사용하는 함수임. 다시 말해, 이 함수는 입력 텐서 **x**와 가중치 텐서사이의 행렬 곱셈을 수행하고, 그 결과에 바이어스를 더하는 연산임.

```

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

```

```

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_()

```

SGD 클래스, 미니배치 확률적 경사 하강법은 모델 파라미터를 최적화하는 알고리즘임.

self.params는 학습할 파라미터들의 리스트들임. **param.grad**는 각 파라미터의 기울기임. **backpropagation**을 통해 계산된 이 **gradient**를 사용하여 **parameter**를 업데이트하게 됨.

zero_grad(self)는 각 파라미터의 기울기를 0으로 초기화하는 역할을 함.

```

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

```

```

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

```

```

@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:
                self.clip_gradients(self.gradient_clip_val, self.model)
        self.optim.step()

```

```

self.train_batch_idx += 1
if self.val_dataloader is None:
    return
self.model.eval()
for batch in self.val_dataloader:
    with torch.no_grad():
        self.model.validation_step(self.prepare_batch(batch))
self.val_batch_idx += 1

```

prepare_batch는 데이터 배치를 준비하는 함수임. fit_epoch(self)는 학습 데이터셋과 검증 데이터셋을 통해 모델을 학습하고, 성능을 평가하는 절차가 들어감.

self.model.train()는 모델을 학습모드로 전환함. 기본적으로 PyTorch에선 train(), eval()을 통해 학습 모드와 평가 모드를 전환할 수 있음. for 부분은 학습 데이터셋에 대해 미니배치 단위로 모델을 학습하는 과정임.

for batch in self.train_dataloader 이 함수는 학습 데이터셋에서 각 배치를 가져옴.

self.model.training_step(self.prepare_batch(batch))는 각 배치에 대해 모델의 training_step메소드를 호출하여 손실을 계산함.

self.optim.zero_grad(): 이전 단계에서 계산된 기울기를 0으로 초기화함.

loss.backward()는 backpropagation을 통해 기울기를 계산함.

if self.gradient_clip_val > 0: self.clip_gradients(self.gradient_clip_val, self.model): 만약 기울기 클리핑 값(gradient_clip_val)이 0보다 크다면, 기울기 클리핑을 수행하여 기울기의 값이 너무 커지지 않도록 제한함.

self.optim.step(): 최적화 알고리즘(예: SGD, Adam)을 통해 모델의 파라미터를 업데이트함.

self.train_batch_idx += 1: 현재 처리한 학습 배치의 인덱스를 증가시킴. 학습 과정에서 배치 인덱스를 추적하는 데 사용됨.

검증 데이터셋을 사용하여 모델을 평가하는 부분임:

self.model.eval(): 모델을 평가 모드로 전환합니다. 평가 모드에서는 드롭아웃 등이 비활성화됨.

for batch in self.val_dataloader: 검증 데이터셋에서 배치를 반복함.

with torch.no_grad(): 검증 과정에서는 기울기 계산이 필요 없으므로 torch.no_grad() 문을 사용하여 역전파를 비활성화함. 이를 통해 메모리와 연산을 절약할 수 있음.

self.model.validation_step(self.prepare_batch(batch)): 각 배치에 대해 모델의 validation_step 메서드를 호출하여 검증 과정을 수행함.

self.val_batch_idx += 1: 현재 처리한 검증 배치의 인덱스를 증가시킴.

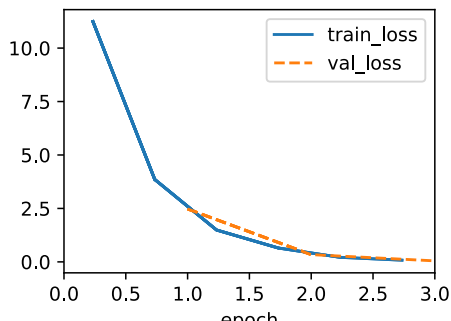
이 코드는 딥러닝 모델의 학습 및 검증 절차를 epoch단위로 처리하는 방법을 보여줌. 1. 학습 모드에서 각 미니배치에 대해 손실을 계산하고, backpropagation을 통해 기울기를 계산하여 parameter를 업데이트 함. 2. 검증 모드에서 검증 데이터셋을 통해 모델을 평가함. 3. 기울기 클리핑, 기울기 초기화를 통해 최적화함.

따라서 모델을 반복적으로 학습하면서 손실을 줄이고, 검증 데이터셋을 이용해 모델의 성능을 평가함.

```

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([ 0.1365, -0.1612])
error in estimating b: tensor([0.2102])

```

3.4 Discussion

모델, loss function, a minibatch stochastic gradient descent optimizer, training function을 모두 구현해보는 챕터임. 이들의 구성요소와 체계를 경험해보는 것이 이 챕터의 목표였음.

먼저 forward메소드를 정의하여 모델을 정의하고, 손실 함수를 정의하고 다음으로 minibatch를 활용하여 매개변수에 대한 손실의 기울기를 추정하며 손실을 줄이는 방향으로 매개변수를 업데이트하는 과정을 거쳤음.

매개변수, 손실 함수, 모델, 최적화 기법이 준비된 후 훈련 루프를 구현하게 됨. 각 훈련에서 반복되는 부분에서는 훈련 예제의 mini batch를 이용하여 모델의 training step 메소드를 통해 손실을 계산함. 이후 각 매개변수에 대한 기울기를 계산하고 최적화 알고리즘을 호출하여 모델 매개변수를 업데이트하게 됨. 요약하면 루프는 다음 과정을 거치게 됨; 매개변수를 초기화 하고, 기울기를 계산하여 이에 따라 매개변수를 업데이트하는 과정을 반복함.

이 세션에서는 데이터를 로드하고, 모델, 손실 함수, 최적화 함수, 시각화의 과정을 테스트 해보았음. 이들을 통해 현대 딥러닝의 기본적인 절차에 대한 감을 익힐 수 있었음.

최적화 기법에는 무엇이 있는지 궁금했음. 다음의 정도로 요약할 수 있었음. SGD: 기본적인지만, 노이즈로 인해 불안정. Momentum: 빠르고 안정적. Adam: 현재 가장 널리 사용되는 최적화 알고리즘, 빠르고 안정적. RMSProp: 순환 신경망에 적합한 알고리즘. Adagrad: 희소 데이터에 적합. Nadam: Nesterov 모멘텀을 활용한 Adam의 변형.

✓ 4.1 Softmax Regression

4.1 Discussion

이 장이 강조하는 점은 제공 오차를 최소화하는 것 말고도 여러 추정 방법이 존재한다는 것임. 더 나아가서는 지도 학습에 회귀만 있는 것이 아님을 나타냄.

분류에 있어 자주 쓰이는 방법 중 하나는 one-hot encoding임. 이들은 여러 카테고리를 지니며, 해당 카테고리에 들 경우 1, 아닐 경우 0을 반환하는 심플한 모델임.

선형 모델을 보았음. 4개의 input layer과, 4종류의 weight를 이용하여 3개의 output layer를 만드는 모습을 볼 수 있었음.

각 출력이 발생할 확률을 구하고, 이들이 양수이면서도 합이 1이 되도록 만들 softmax 기법에 대해 학습함. 이들은 주어진 각 클래스의 조건부 확률이라고 할 수 있음. 이들과 실제를 비교함으로써, 추정치를 현실과 비교할 수 있음.

추가로 엔트로피를 활용해서 실제 확률과 계산된 확률간의 차이에 대한 체감을 가능케 함. 다시 말해 예상된 행동과 그 예측 사이의 차이를 얻을 수 있음.

softmax말고도 이러한 출력의 확률을 구할 방법에 대해 알아보았음. 이 중 대표적인 방법은 Sigmoid 함수, Temperature Scaling, 베이지안 딥러닝 같은 기법들이 있으며, 모델의 불확실성을 해석할 때는 MC-Dropout이나 엔트로피 분석 같은 방법이 유용할 수 있다는 것을 알게 됨.

✓ 4.2 The Image Classification Dataset

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

d2l.use_svg_display()

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


배치의 크기를 64로 설정함. 다음으로 이미지 크기를 조정하는데 기본값을 (28, 28)로 설정함.

`transforms.Compose`는 여러 이미지 변환 작업을 연속으로 적용할 수 있도록 묶어주는 함수임.

`transforms.Resize(resize)`는 이미지 크기를 `resize`로 조정함. 이미지를 28*28 픽셀로 변환함.

`transforms.ToTensor()`는 이미지를 PyTorch 텐서로 변환하는 것임.

`torchvision.datasets.FashionMNIST`: Fashion-MNIST 데이터셋을 로드하는 함수임. `root=self.root`: 데이터를 저장할 디렉토리 경로를 설정합니다. 이 값은 상위 클래스인 `d2l.DataModule`에서 제공하는 `self.root`로 설정됨. `train=True`: 학습용 데이터셋을 로드함. Fashion-MNIST는 학습용과 검증용으로 나뉘어 있는데, `train=True`는 학습용 데이터를 의미함. `transform=trans`: 위에서 정의한 이미지 전처리(`transforms`)를 적용함. 이미지 크기를 변경하고 텐서로 변환하는 작업을 수행함. `download=True`: 데이터셋이 로컬에 없을 경우 자동으로 다운로드함.

이 부분은 검증(validation) 데이터셋을 로드하는 과정임. `train=False`: 검증용 데이터셋을 로드함. `train=False`는 검증 또는 테스트용 데이터를 의미함. `transform=trans`: 학습 데이터셋과 동일하게 이미지 전처리 작업을 적용함. `download=True`: 검증용 데이터셋이 로컬에 없을 경우 자동으로 다운로드함.

이 코드는 PyTorch와 torchvision을 이용하여 Fashion-MNIST 데이터셋을 로드하고, 학습 및 검증 데이터셋에 대해 필요한 전처리 작업을 수행하는 클래스를 정의한 것임.

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

```
↗ (60000, 10000)
```

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

```
↗ torch.Size([1, 32, 32])
```

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

```
@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:557: UserWarning: This DataLoader will create 4 worker processes
  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'
```

```
↗ /usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 worker processes
  warnings.warn(_create_warning_msg(
'12.83 sec'
```




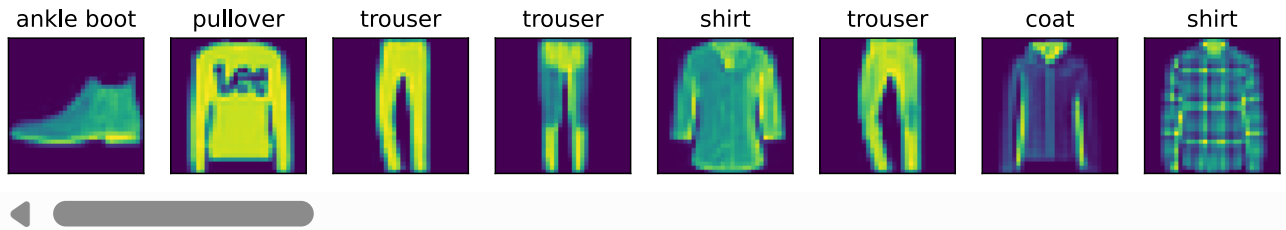
```
def show_images(imgs, num_rows, num_cols, titles = None, scale=1.5):
    """Plot a list of images."""
    raise NotImplementedError
```

```
@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)
```

 /usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 worker processes with 2 GPUs each. Using a 2 GPUs machine for training is recommended. If 2 GPUs are not available, workers on the same GPU will have to share memory. This will increase the memory usage of each worker and potentially result in a $RuntimeError$ if the GPU has insufficient memory. For sharing memory, use `worker_init_fn` to reduce the `pin_memory` size.



- 여기서 next는 iteration의 다음 값을 반환함

4.2 Discussion

이 장에서는 Fashion_MNIST dataset을 이용한 Image Classification을 다룸.

이들을 len과 chw 텐서로 저장한 값들 또한 확인해볼 수 있었음. 이 데이터를 불러와서 적절히 학습시킨 후, 시각화 하면 이미지가 분류되는 모습을 볼 수 있었음.

이들을 batch size, number of channels, height, width 등으로 일반화하는 모습을 볼 수 있었음.

문득 이미지를 분류하는 걸 응용하는 분야는 무엇이 있나 생각해보았음. 알아본 바로는 자동화, 사용자 맞춤형 경험 제공, 생산성 향상, 보안 강화 등의 이점을 제공하며, 이는 현재 전자 상거래, 의료, 자율 주행, 보안, 제조, 농업 등에서 핵심 기술로 쓰이고 있다는 것을 알 수 있었음.

4.3 The Base Classification Model

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

```
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)
```

```
@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
```

Y_{hat} 은 예측된 값들, Y 는 실제 정답 값임. $averaged = True$ 는 평균을 낼지 결정하는 플래그임.

$Y_{hat} = Y_{hat}.reshape(-1, Y_{hat}.shape[-1])$ 이 부분은 Y_{hat} 의 크기를 마지막 차원만 남기고 평탄화하는 것임. 다시말해, 예측값이 여러 차원의 경우에도 계산가능하도록 배열을 2차원으로 변경하는 것임.

$Y_{hat}.argmax(axis=1)$ 는 예측된 값중 가장 높은 값을 가진 클래스 인덱스를 가져옴. 즉, 각 샘플에 대해 모델이 예측한 클래스가 무엇인지를 나타냄. $type(Y.dtype)$ 은 예측값의 데이터타입을 실제 정답과 동일하게 맞춰줌.

compare 부분은 예측값과 실제 정답을 비교하여 같은지를 판단하는 것임. Y 의 차원도 1차원으로 변환하고, 예측이 맞으면 True, 아니면 False를 반환하도록 하는 것임.

4.3 Discussion

validation step 메소드에서는 loss function을 통해 loss 값과 분류 정확도를 모두 보고함. class로 나누고 나서 이에 대한 accuracy를 추정하기도 함. 예측이 레이블 클래스 y와 일치할 때, 이것을 정확한 예측이라고 함. 여기서 분류 정확도는 모든 예측 중에서 정확한 예측의 비율임. 정확도는 성능 측정 지표가 될 수 있음.

모델의 성능을 측정하기 위해 '정확도'를 판단한다는 것을 알 수 있었음.

4.4 Softmax Regression Implementation from Scratch

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

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

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

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition # The broadcasting mechanism is applied here
```

softmax를 직접 함수로 구현한 것임

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

```
↔ (tensor([[0.1714, 0.1563, 0.2959, 0.1639, 0.2125],
           [0.2049, 0.2285, 0.2471, 0.1239, 0.1956]]),
    tensor([1., 1.]))
```

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

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

num_inputs: 입력 변수의 수 (특성 수)를 나타냄. num_outputs: 출력 변수의 수 (클래스 수)를 나타냄. lr: 학습률을 설정하는 변수임. sigma: 가중치 초기화 시 사용하는 표준편차임. 기본값은 0.01로 설정되어 있음.

self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs), requires_grad=True) 는 평균이 0이고 표준편차가 sigma인 정규분포를 따르는 난수를 생성하는 함수임.

self.W는 소프트맥스 회귀의 가중치 행렬로, 차원은 (num_inputs, num_outputs)임. 즉, 입력 변수 수만큼의 행과 출력 변수 수만큼의 열을 갖는 2차원 배열(행렬)임.

requires_grad=True는 이 가중치가 학습 중에 기울기를 계산할 수 있도록 설정된다는 의미임.

self.b는 편향(bias)임. 출력 변수 수만큼의 1차원 벡터로, 모든 값이 0으로 초기화 됨.

결국 이 코드는 softmax 회귀 모델을 구현한 클래스임. 이 클래스는 가중치(w)와 편향(b)를 초기화하며, 학습할 때는 parameters 메소드를 통해 이 두 파라미터를 최적화함.

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

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

```
tensor([0.1000, 0.5000])
```

cross-entropy loss function을 구현함. 이는 모든 딥러닝의 loss function에서 가장 기초가 되는 것임. 3개의 클래스에 대한 두 가지 예측 확률로 구성된 샘플 데이터와 레이블을 이용, 선택된 확률의 로그를 평균내어 이를 구현해보았음.

```
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)
```

list(range(len(y_hat)))는 y_hat에서 각 샘플에 대한 인덱스를 얻기 위해 사용됨. 예를 들어 y_hat의 길이가 4인 경우 list(range(len(y_hat)))는 [0, 1, 2, 3]과 같은 리스트가 됨.

(y_hat[list(range(len(y_hat))), y])는 y_hat에서 각 샘플에 대해 실제 정답 클래스의 확률 값을 추출함.

torch.log()는 선택된 정답 클래스에 대한 확률 값들에 로그 값을 취함. 이는 교차 엔트로피 손실 함수의 수학적 정의 때문임. 이 함수는 모델이 낮은 확률을 할당한 정답에 대해 큰 페널티를 부과하도록 설계됨.

이 로그값에 음수를 취하여 손실을 계산하고, mean을 통해 모든 샘플에 대한 손실의 평균을 구함. 이 값이 최종 손실 값이 됨.

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

훈련하기 위해, fit 메소드를 활용함. max-epochs, batch size, lr의 하이퍼파라미터를 조정하여 학습시킴. 추가로 Fashion-MNIST의 테스트 데이터를 검증 셋으로 두고, 이에 대한 loss, accuracy를 측정하는 함수를 만들었음.

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



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

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create 4 worker processes  
warnings.warn(_create_warning_msg(  
torch.Size([256])
```

```
wrong = preds.type(y.dtype) != y  
X, y, preds = X[wrong], y[wrong], preds[wrong]  
labels = [a+]
```

잘못 분류한 이미지를 파악하기 위해, 실제 레이블과 예측 값을 비교하여 시각화하는 과정을 거치기도 했음.

4.4 Discussion

softmax 회귀는 가장 기초적임. softmax 공식은 확률을 측정하고 싶은 원소의 exp값을, 각 원소의 exp값들의 합으로 나누면 됨.

다루게 될 raw data 2828 pixel image를 1784로 취급하는 것이 시작임. dataset이 10개의 클래스들을 갖기에 10784에 biases들을 위해 110를 추가하는 것으로 초기화 할 수 있음. 이를 함수로 구현했음. biases를 위해 얼마를 추가하는지에 대해 척도를 알고 싶었음.

일반적으로 0으로 초기화하거나, 작은 랜덤 값으로 초기화함. 이후 가중치와 마찬가지로, 역전파를 통해 손실 함수의 기울기에 따라 학습됨. 이러한 Bias는 활성화 함수와 결합해 비선형성을 추가하여 신경망의 표현력을 향상시킴.

5.1. Multilayer Perceptrons

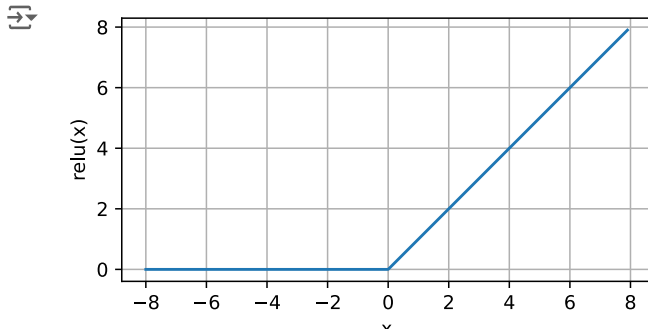
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

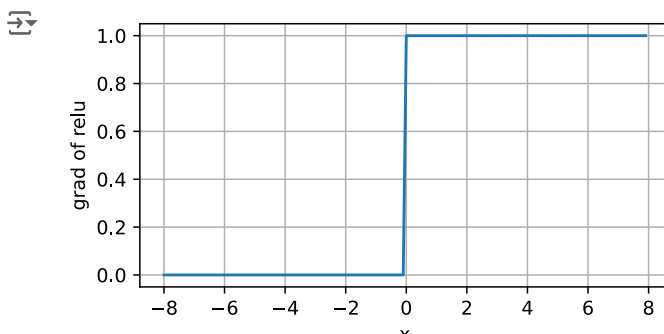
$$\text{ReLU}(x) = \max(x, 0)$$

relu는 간단하고, 다양한 예측 작업에서 좋은 성능을 제공함. 이는 양수의 요소만 유지하고, 음수 요소는 0으로 설정해버려 폐기해버림. ReLU에 선형 항을 추가한 pReLU가 사용되기도 함. 이는 음수 요소의 일부가 전달되도록 기여함.

```
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))
```

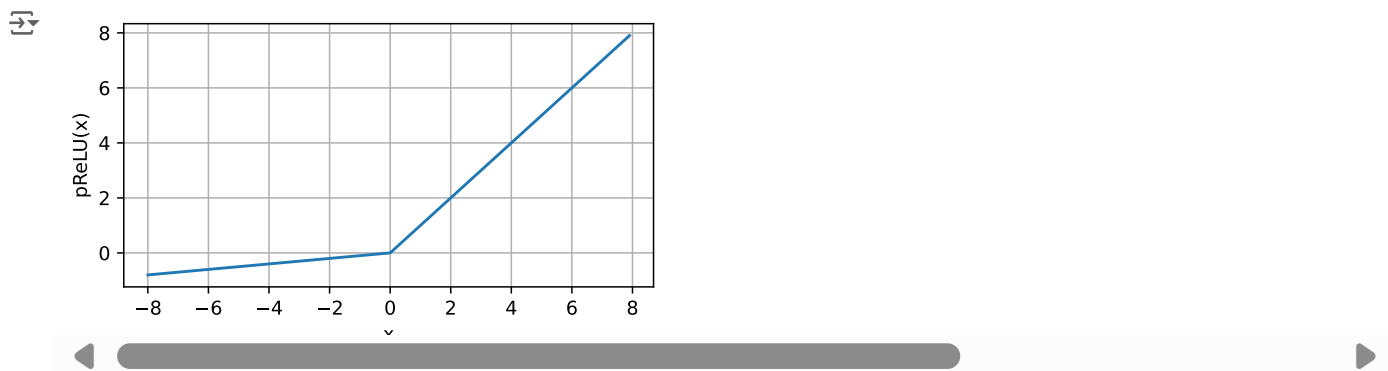


```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```

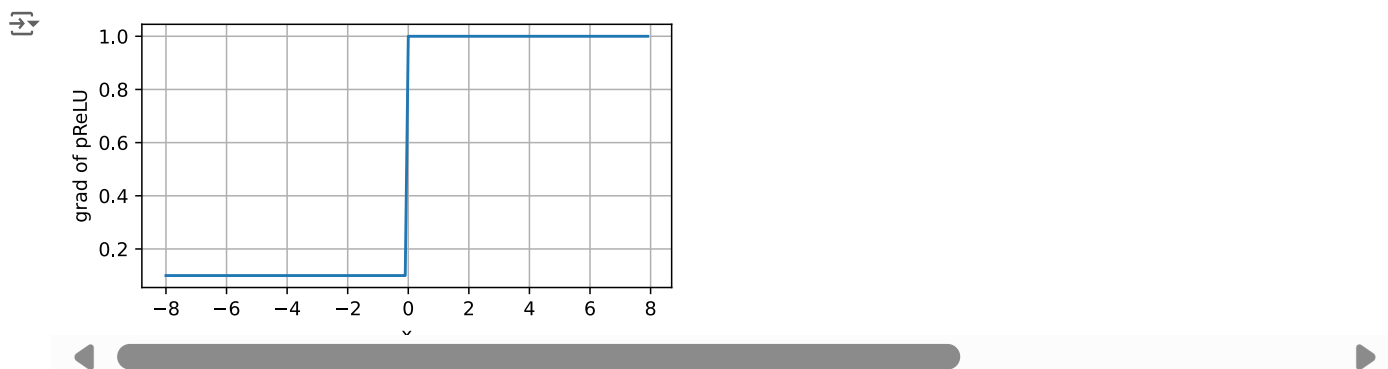


```
pReLU = lambda x, a: torch.max(torch.tensor(0), x) + a * torch.min(torch.tensor(0), x)
```

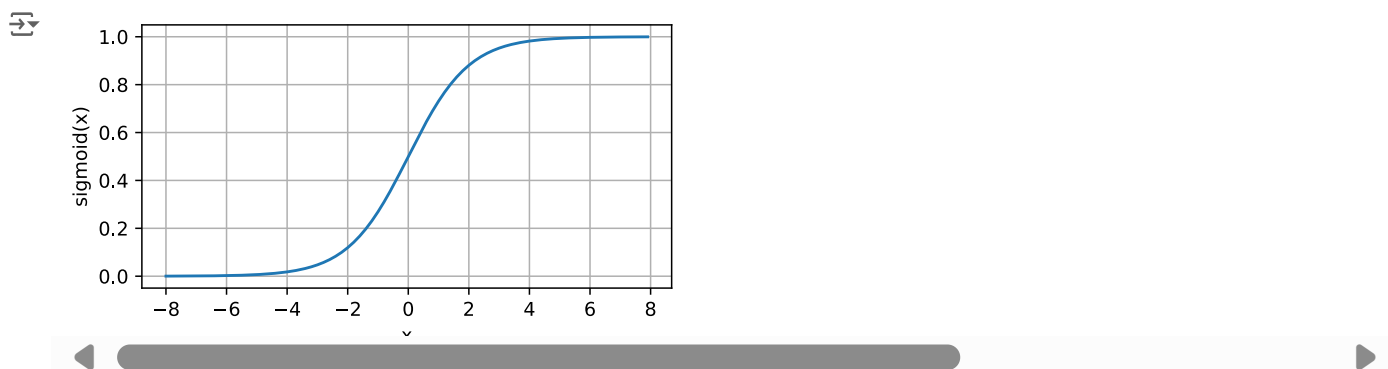
```
y = pReLU(x=x, a = 0.1)
d2l.plot(x.detach(), y.detach(), 'x', 'pReLU(x)', figsize=(5, 2.5))
```



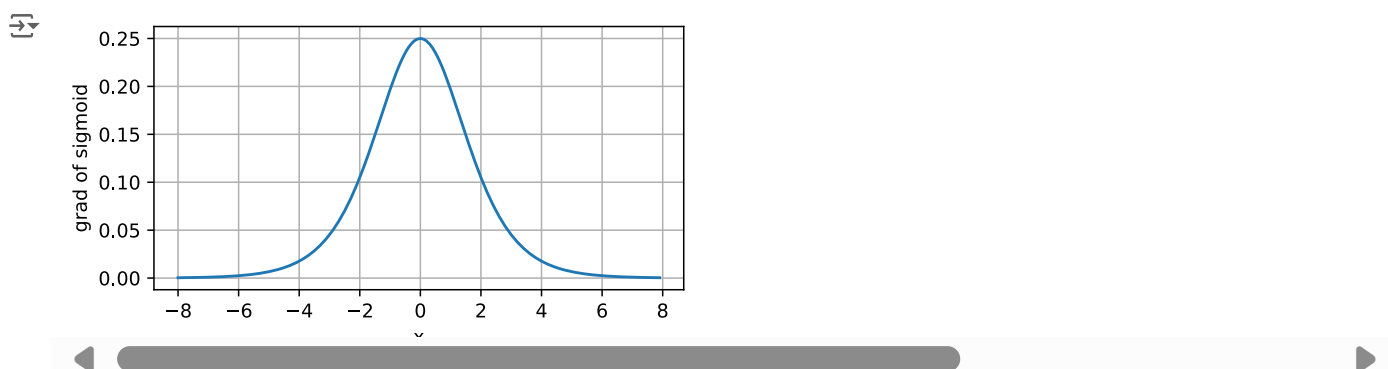
```
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of pReLU', figsize=(5, 2.5))
```



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

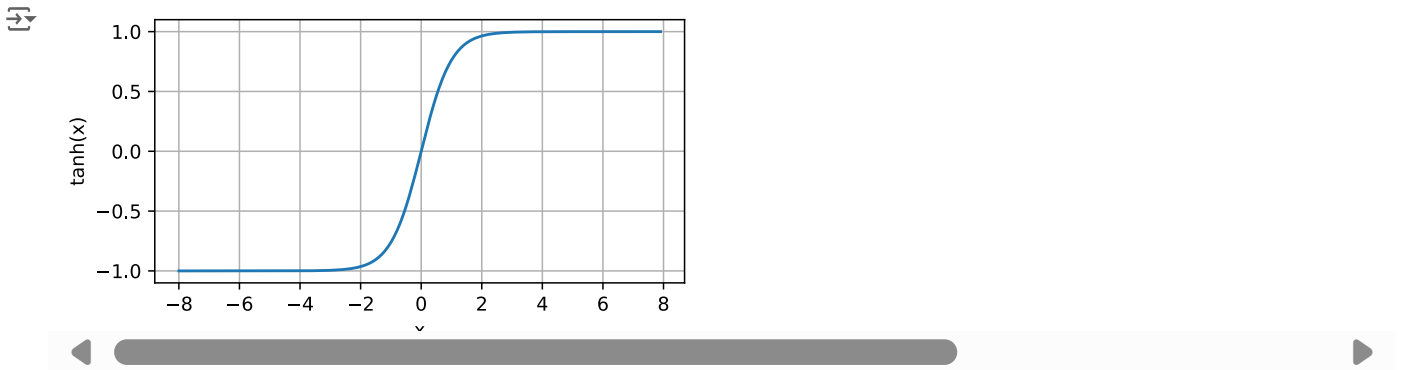


```
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



•
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

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



5.1 Discussion

보다 더 심화된 deep neural networks를 위해 선형 모델의 한계를 이해함. 선형 모델의 한계는 단조성임. 어떤 특징이 증가하면, 가중치가 양수인 경우에는 모델 출력이 항상 증가하고, 음수인 경우에는 항상 감소한다는 한계를 지님. 가령, 고양이와 개를 분류할 때, 픽셀 강도가 증가한다고 해서, 해당 그림이 고양이일 가능성 혹은 개일 가능성이 올라가는 것은 아님.

이를 해결하기 위해 비 선형성 모델을 도입하는 것임. hidden layers를 통해 해당 문제에 해결 실마리를 찾을 수 있음. 가장 간단한 해결책은 fully connected layer를 여러 개 쌓는 것임. 이를 MLP라고 함. 단순히 hidden layers를 추가하면 보다 표현력이 높은 모델을 얻을 수 있음. Cybenko(1989)의 연구에서 단일 hidden layer가 충분한 노드를 갖추고 있고, 적절한 가중치를 제공한다면 어떤 함수도 모델링 할 수 있다는 것을 밝혀냈음. 그러나 이는 매우 어려운 일임. Activation Function을 활용하는 것이 보다 더 효과적, 정확하게 해결할 수 있음.

가장 보편적인 비선형 변환, Activation Function은 ReLU임.

sigmoid는 입력값을 0과 1 사이의 출력으로 변환함. 이는 입력을 음의 무한대 - 양의 무한대로 받으면, 출력을 0,1 범위로 압축하는 효과를 지님. 이는 sigmoid가 특정 임계값을 넘으면 1 혹은 0을 반환하도록 설정되었기 때문임. 물론 최적화 측면에서 sigmoid의 한계가 존재하기 때문에 대부분은 ReLU로 대체되었음.

tanh 함수는 입력값이 0에 가까울 때, 선형 변환에 가까운 모습을 출력함. 함수의 모양이 sigmoid와 비슷하지만, tanh는 좌표계의 원점에서 점 대칭성을 보임.

비선형성을 도입하면 표현력이 보다 높은 다층 신경망 아키텍처를 구축할 수 있음. 여기서 나온 ReLU, Sigmoid, Tanh중 최적화에 가장 적합한 모델은 ReLU임. 대부분의 학자들은 ReLU를 사용할까?

일단 ReLU는 대부분의 딥러닝 모델에서 활성화 함수로 널리 사용됨. 비선형성, Gradient Vanishing 문제 완화, 계산 효율성, 희소 활성화 등의 장점으로 인해 주로 사용됨. 물론 ReLU의 가장 큰 문제 중 하나인 Dead ReLU 문제를 해결하기 위해 Leaky ReLU, PReLU, ELU 같은 변형된 활성화 함수들이 개발되었음. 그러나 다른 activation function이 더 적합한 경우도 있음.

다음의 경우에 각각의 activation function을 사용함

Sigmoid: 이진 분류 문제에서 출력층에 많이 사용됨. 값이 [0, 1] 범위로 제한되어 확률처럼 해석할 수 있기 때문임.

Tanh: 중간 레이어에서 사용할 때 Sigmoid보다 선호되며, [-1, 1] 범위에서 출력을 제공하여 데이터가 중심에서 퍼지게 만듦.

Softmax: 다중 클래스 분류 문제에서 출력층에 사용됨. 각 클래스에 대한 확률 분포를 제공함.

5.2 Implementation of Multilayer Perceptrons

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

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))
```

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

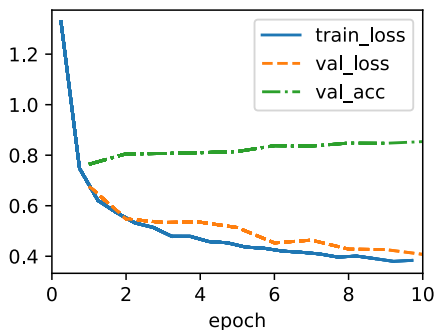
ReLU가 함수로 직접 구현되었음.

```
@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
```

2차원 이미지를 길이가 num_inputs인 1차원 벡터로 변형하였음.

이를 model, data, trainer가 정의 된 fit method를 통해 학습시킴.

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



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

nn.Flatten():

입력 데이터를 1차원 벡터로 변환하는 레이어임. 예를 들어, 이미지와 같이 2D 또는 3D 형태의 데이터를 다룰 때, 이를 1차원으로 변환하여 MLP에 입력할 수 있도록 함. 28x28 크기의 이미지를 입력 받으면, 이를 (28*28)인 784개의 특성으로 펼쳐줍니다.

nn.LazyLinear(num_hiddens):

LazyLinear는 레이어가 처음 데이터를 받을 때 입력 크기를 자동으로 결정하는 선형(fully connected) 레이어임. 즉, 이 레이어는 처음 입력이 주어질 때 입력 차원을 자동으로 추론함.

num_hiddens는 hidden layer의 뉴런 수를 지정하는 파라미터임. 이 레이어는 입력 데이터를 hiddenlayer 차원으로 변환함.

nn.ReLU():

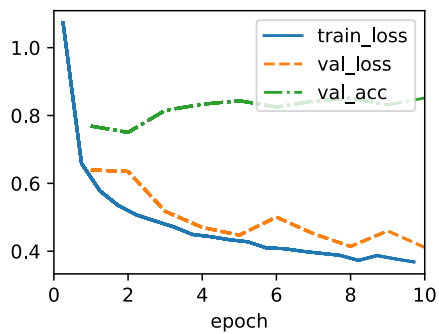
ReLU(Rectified Linear Unit)는 비선형 활성화 함수로, 각 뉴런의 출력 값이 0보다 작을 경우 0을 반환하고, 그렇지 않으면 그대로 반환함. 이 비선형성을 통해 모델이 복잡한 패턴을 학습할 수 있게 해줌.

nn.LazyLinear(num_outputs):

마지막 레이어는 출력 차원을 결정하는 Linear 레이어임. num_outputs는 최종 출력의 차원을 지정함. 예를 들어, 이 모델이 분류 문제를 푼다면, num_outputs는 예측하려는 클래스 수가 됨. 마찬가지로 LazyLinear이므로 처음 입력을 받을 때 입력 차원을 자동으로 추론하게 됨.

정리하면, MLP는 입력 데이터를 순차적으로 처리함. Flatten을 통해 입력 데이터를 1차원 벡터로 펼침. 첫 번째 LazyLinear를 통해 입력 데이터를 hidden layer의 뉴런 수로 변환하고, ReLU를 통해, hidden layer의 출력을 변환함. 두 번째 LazyLinear로 은닉층의 출력을 최종 출력 차원으로 변환하여 최종 예측 값을 생성함.

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

✓ 5.2 Discussion

해당 챕터에서 하나의 hidden layer와 256개의 hidden units을 가진 MLP를 구현하게 됨. 이와 같은 층의 수와 너비는 구현자가 조정 가능한 hyperParameter임. 일반적으로 층의 너비는 2의 제곱수로 나누어질 수 있도록 선택함. 이는 하드웨어에서 메모리가 할당되고, 주소화되는 방식을 고려할 때, 계산에 효율적이기 때문임.

각 층마다 하나의 가중치 행렬과 편향 벡터를 파악하며 코드가 작성되었음.

이 챕터에서 기존의 모델에 추가된 것은 이전 챕터에서는 한 개의 fully connected layer를 추가했던 것과 달리, 두 개의 fully connected layer를 추가하였음. 여기서 첫 째는 hidden layer이고, 둘 째는 output layer임. 모델의 parameter를 입력으로 받아 이를 변형하는 forward 메소드 대신, 여기서는 sequential class에서 forward 메소드를 상속받아 구현됨. 이를 훈련 루프에 넣고, 훈련 시킴.

이 챕터를 통해 single layer에서 multilayer로 확장되는 경험을 할 수 있음. 물론 여러 통계적 기본 사항과 효율적인 모델의 계산은 여전히 과제임.

✓ 5.3 Forward Propagation, Backward Propagation, and Computational Graphs

5.3 Discussion

딥러닝의 상호작용에 대한 이해가 들어가 있음. 지금까지 forward propagation 계산에 대해서 주로 주목하였지만, 딥러닝을 정확히 이해하기 위해서는 기울기가 어떻게 계산되는지, 즉 backpropagation 동작을 이해해야 함.

J를 목적 함수로 두고, 이를 바탕으로 forward propagation 계산을 행할 수 있음. 그렇다면 backpropagation은 신경망 parameter의 기울기를 계산하는 것임. 출력에서 입력층으로 네트워크를 역방향 탐색하는 것임.