

✓ COSE474-2024F: Deep Learning HW1

✓ 0.1 Installation

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



Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel)

Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (from ipykernel)

Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipykernel)

Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel)

Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel)

Requirement already satisfied: prompt-toolkit!=3.0.0,!3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel)

Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from jupyter-client)

Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: Jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter-client)

Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packages (from jupyter-server>=1.10.0)

Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=4.10.0)

Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/python3.10/dist-packages (from jupyter-server>=1.10.0)

Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from argon2-cffi>=21.3 in /usr/local/lib/python3.10/dist-packages)

Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0.1 in /usr/local/lib/python3.10/dist-packages)

Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages (from jupyter-server>=1.10.0)

Requirement already satisfied: websocket-client in /usr/local/lib/python3.10/dist-packages (from jupyter-server>=1.10.0)

Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages)

Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/dist-packages (from anyio<4,>=3.1.0 in /usr/local/lib/python3.10/dist-packages)

✓ 2.1. Data Manipulation

```
1 import torch
```

```
1 x = torch.arange(12, dtype=torch.float32)
2 x
```

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

```
1 x.numel() # number of element of the tensor
```

```
⇒ 12
```

```
1 x.shape # length along each axis
```

```
⇒ torch.Size([12])
```

```
1 X = x.reshape(3,4) # change the shape of a tensor n*m matrix → 3 is row 4 is column
2 X
```

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

```
1 A = x.reshape(3, -1) # 추론되어야 할 구성 요소에 -1을 지정
2 A
```

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

```
1 torch.zeros((2, 3, 4)) # 2개짜리 3x4 matrix .. 0으로
```

```
⇒ 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.]]]])
```

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

1 torch.ones((2, 3, 4)) # 2개짜리 3x4 matrix .. 1로

```
⇒ 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.]])])
```

1 torch.randn(3, 4) # 평균이 0이고 표준 편차가 1인 표준 가우시안(정규) 분포에서 요소를 추출

```
⇒ tensor([[ 0.9148, -1.1175, -0.5454,  1.2875],
          [-0.7733,  1.0141, -0.9207, -1.0154],
          [ 0.1585,  0.6872,  0.5382, -0.2337]])
```

1 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]])
```

1 X[-1], X[1:3] # 리스트 끝에서부터 접근하려면 음수 인덱싱

2 # X[-1] → 인덱스 1개 → 0번째 축. 마지막 행

3 #X[1:3] → 두번째와 3번째 row

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

1 X[1,2] = 17 # rewrite elements

2 X

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

1 X[:2, :] = 12 # assign multiple elements

2 # :2는 1번째 2번째 row 선택, :는 column 전

3 X

```
⇒ tensor([[12., 12., 12., 12.],
          [12., 12., 12., 12.],
          [ 8.,  9., 10., 11.]])
```

1 x

```
⇒ tensor([12., 12., 12., 12., 12., 12., 12., 12.,  8.,  9., 10., 11.]])
```

```
1 torch.exp(x) # elementwise operation → 텐서의 각 요소에 표준 스칼라 연산을 적용
2 # unary scalar operation.. exp는 지수함수
```

```
→ 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])
```

```
1 x = torch.tensor([1.0, 2, 4, 8])
2 y = torch.tensor([2, 2, 2, 2])
3 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.]))
```

```
1 X = torch.arange(12, dtype=torch.float32).reshape((3,4))
2 Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
3 X, Y, torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
4 # tensor 2개를 dim=0으로 결합 → row 개수가 늘어남
5 # tensor 2개를 dim=1로 결합 → column 개수가 늘어남
6 #axis 0 → row / axis 1 → column
```

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

```
1 X, Y, X==Y
```

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

```
1 X, X.sum() # summing all the elements in the tensor
```

```
→ (tensor([[ 0.,  1.,  2.,  3.],
           [ 4.,  5.,  6.,  7.],
           [ 8.,  9., 10., 11.]]),
   tensor(66.))
```

```
1 a = torch.arange(3).reshape((3, 1))
2 b = torch.arange(2).reshape((1, 2))
3 a, b
```

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

```
1 a + b
2 # broadcasting ; 서로 다른 크기의 행렬의 연산 → 작은 행렬의 차원을 자동으로 확장
3 # a는 3x1, b는 1x2 → a는 column 방향으로, b는 row 방향으로 확장
```

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

```
1 before = id(Y) # id함수: 객체의 고유한 메모리 주소 반환
2 Y = Y + X      # 새로운 메모리 할당받음
3 id(Y) == before
```

```
→ False
```

```
1 Z = torch.zeros_like(Y) # 텐서 Y와 동일한 모양과 크기를 가지지만 모든 값이 0으로 채워진 텐서 Z를 생성
2 print('id(Z):', id(Z)) # 텐서 Z의 메모리 주소를 출력
3 Z[:] = X + Y # 슬라이스 표기법 → 텐서 Z 자체의 메모리 주소는 변경 X
4 print('id(Z):', id(Z)) # 동일한 메모리 주소 출력
```

```
→ id(Z): 138164937555488
   id(Z): 138164937555488
```


```
1 before = id(X)
2 X += Y
3 id(X) == before
```

```
→ True
```

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

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


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

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

✓ 2.2. Data Preprocessing


```
1 import os
2
3 os.makedirs(os.path.join('.', 'data'), exist_ok=True)
4 data_file = os.path.join('.', 'data', 'house_tiny.csv')
5 with open(data_file, 'w') as f:
6     f.write("""NumRooms,RoofType,Price
7 NA,NA,127500
8 2,NA,106000
9 4,Slate,178100
10 NA,NA,140000""")
```

```
1 import pandas as pd
2
3 data = pd.read_csv(data_file)
4 print(data)
```

 NumRooms RoofType Price


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

```
1 # train models to predict a designated target value
2 # NaN은 imputation (추정치로 채우기) / deletion (NaN이 있는 row, column 제거) 으로 처리 가능
3 inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2] # 우선 separate out columns to input versus target values
4 inputs = pd.get_dummies(inputs, dummy_na=True) #input → NaN 값 카테고리 처리
5 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

```
1 inputs = inputs.fillna(inputs.mean()) # replace NaN entries with the mean value of corresponding column
2 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

```
1 import torch
2
3 X = torch.tensor(inputs.to_numpy(dtype=float))
```

```
4 y = torch.tensor(targets. to_numpy(dtype=float))
```

```
5 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))
```

✓ 2.3. Linear Algebra

```
1 import torch
```

```
1 x = torch.tensor(3.0)
```

```
2 y = torch.tensor(2.0)
```

```
3
```

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

```
⇒ (tensor(3.), tensor(2.), tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

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

```
2 x
```

```
⇒ tensor([0, 1, 2])
```

```
1 x[2]
```

```
⇒ tensor(2)
```

```
1 len(x)
```

```
⇒ 3
```

```
1 x.shape
```

```
⇒ torch.Size([3])
```

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

```
2 A
```

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

```
1 A.T # transpose
```

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

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

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

```
1 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]]])
```

```
1 A = torch.arange(6, dtype=torch.float32).reshape(2,3)
2 B = A.clone()
3 A, A + B
4
```

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

```
1 A * B
```

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

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

```
⇒ (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]]]),
   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]))
```



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

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

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

```
⇒ (tensor([[0., 1., 2.],
           [3., 4., 5.]]),
    torch.Size([2, 3]),
    tensor(15.))
```

```
1 A, A.shape, A.sum(axis=0).shape, A.sum(axis=0) # axis = 0 → row를 따라 연산
```

```
⇒ (tensor([[0., 1., 2.],
           [3., 4., 5.]]),
    torch.Size([2, 3]),
    torch.Size([3]),
    tensor([3., 5., 7.]))
```

```
1 A, A.shape, A.sum(axis=1).shape, A.sum(axis=1) # axis = 1 → column을 따라 연
```

```
⇒ (tensor([[0., 1., 2.],
           [3., 4., 5.]]),
    torch.Size([2, 3]),
    torch.Size([2]),
    tensor([ 3., 12.]))
```

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

```
⇒ tensor(True)
```

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

```
⇒ (tensor(2.5000), tensor(2.5000))
```

```
1 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]))
```

```
1 A.mean(axis=1), A.sum(axis=1) / A.shape[1]
```

```
⇒ (tensor([1., 4.]), tensor([1., 4.]))
```

```
1 sum_A = A.sum(axis=1, keepdims=True) # sum 계산 이후에도 차원 유지
```

```
2 A, sum_A, sum_A.shape
```

```
⇒ (tensor([[0., 1., 2.],
           [3., 4., 5.]]),
    tensor([[ 3.],
           [12.]]),
    torch.Size([2, 1]))
```

1 A / sum_A # 각 행의 합이 1 (A의 각 요소를 그 행의 합으로 나눔 → 각 행이 합해서 1)

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

1 A, A.cumsum(axis=0) # cumsum : 주어진 axis를 따라 각 요소의 누적합 계산

```
⇒ (tensor([[0., 1., 2.],
          [3., 4., 5.]]),
    tensor([[0., 1., 2.],
          [3., 5., 7.])))
```

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

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

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

1 torch.sum(x*y) # dot product of x and y

```
⇒ tensor(3.)
```

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

2 # 0번째 row와 x의 dot product: $0 \times 0 + 1 \times 1 + 2 \times 2 = 5$

3 # 0번째 row와 x의 dot product: $3 \times 0 + 4 \times 1 + 5 \times 2 = 14$

```
⇒ (tensor([[0., 1., 2.],
          [3., 4., 5.]]),
    tensor([0., 1., 2.]),
    torch.Size([2, 3]),
    torch.Size([3]),
    tensor([ 5., 14.]),
    tensor([ 5., 14.])))
```

1 B = torch.ones(3, 4)

2 A, B, torch.mm(A, B), A@B

```
⇒ (tensor([[0., 1., 2.],
          [3., 4., 5.]]),
    tensor([[1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.]]),
    tensor([[ 3.,  3.,  3.,  3.],
          [12., 12., 12., 12.]]),
    tensor([[ 3.,  3.,  3.,  3.],
          [12., 12., 12., 12.])))
```

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

2 u, torch.norm(u) # norm calculates the l2 norm (euclidean distance)

```
⇒ (tensor([ 3., -4.]), tensor(5.))
```

```
1 torch.abs(u).sum()      #l1 norm → manhattan distance
```

```
⇒ tensor(7.)
```

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

```
2 #모든 element가 1인 4x9 tensor의 l2 norm
```

```
⇒ tensor(6.)
```

✓ 2.5. Automatic Differentiation

```
1 import torch
```

$y = 2\mathbf{x}^\top \mathbf{x}$ 미분할 예정

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

```
2 x
```

```
⇒ tensor([0., 1., 2., 3.])
```

```
1 # y를 벡터 x에 대해 미분할 때, 결과인 gradient를 저장할 공간이 필요
```

```
2 # 같은 parameter에 대해 여러번 미분해야 → 기존 메모리 재사용
```

```
3 x.requires_grad_(True)
```

```
4 x.grad
```

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

```
2 y
```

```
⇒ tensor(28., grad_fn=<MulBackward0>)
```

```
1 y.backward() #y를 x에 대해 미분하려면 backward 메서드 호출하면 됨
```

```
2 #y.backward()→ y에 대한 gradient 계
```

```
3 x.grad
```

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

```
1 x.grad == 4 * x  # 미분 확인
```

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

```
1 x.grad.zero_() # gradient 리셋
```

```
2 y = x.sum()
```

```
3 y.backward()
```

```
4 x.grad
```

```
5
```

```
6 #gradient를 새로 계산해도 기존에 저장된 그래디언트 값이 자동 초기화 X
```

```
7 #→ 초기화해야!!
```

→ tensor([1., 1., 1., 1.])

```
1 x.grad.zero_()
2 y = x * x
3 y.backward(gradient=torch.ones(len(y)))
4 # gradient=torch.ones(len(y))는 자코비안과 곱해질 벡터 → 각요소가 1,, 자코비안의 합과 같은 의미
5 x.grad
```

→ tensor([0., 2., 4., 6.])

```
1 x.grad.zero_()
2 y = x * x
3 u = y.detach()
4 z = u * x # u는 detach, z는 x와만 관련. u는 상수취급
5
6 z.sum().backward()
7 x.grad == u
```

→ tensor([True, True, True, True])

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

→ tensor([True, True, True, True])

```
1 #automatic differentiation (자동미분)을 위한 반복문
2
3 def f(a):
4     b = a * 2
5     while b.norm() < 1000:
6         b = b * 2
7         if b.sum() > 0:
8             c = b
9         else:
10            c = 100 * b
11     return c
```

```
1 a = torch.randn(size=(), requires_grad=True) #requires_grad=True: 자동미분 활성화
2 d = f(a) #f(a)는 linear function → 입력 a에 대해 piecewise defined scale
3 d.backward()
```

```
1 a.grad == d / a
2 #자동 미분을 통해 계산된 f(a)의 gradient는 f(a)/a와 일치해야
```

→ tensor(False)

✓ 3.1. Linear Regression

```

1 %matplotlib inline
2 import math
3 import time
4 import numpy as np
5 import torch
6 from d2l import torch as d2l

```

```

1 # train model 상황
2 n = 10000
3 a = torch.ones(n)
4 b = torch.ones(n)

```

```

1 #for-loops 사용해서 벡터의 각 요소 더하기
2 c = torch.zeros(n)
3 t = time.time()
4 for i in range(n):
5     c[i] = a[i] + b[i]
6 f'{time.time() - t:.5f} sec'

```

→ '0.26363 sec'

```

1 # Vectorization을 사용해 + 호출로 벡터 더하기 → 속도 빠름 + 효율성 최고
2 t = time.time()
3 d = a + b
4 f'{time.time() - t:.5f} sec'

```

→ '0.00024 sec'

```

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

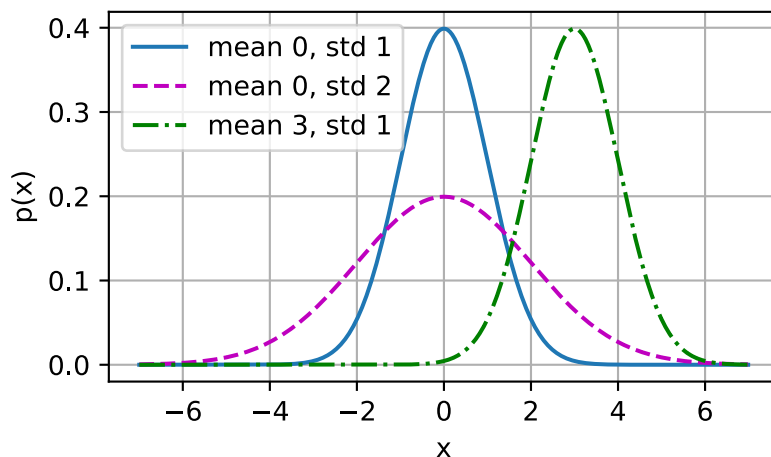
```

```

1 x = np.arange(-7, 7, 0.01)
2 params = [(0, 1), (0, 2), (3, 1)]
3 d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x', ylabel='p(x)', figsize=(4.5, 2.5), legend

```

→



✓ 3.2. Object-Oriented Design for Implementation

```
1 import time
2 import numpy as np
3 import torch
4 from torch import nn
5 from d2l import torch as d2l
```

```
1 def add_to_class(Class): # 동적으로 메서드 추가 (클래스 생성한 후에도 그 클래스에 메서드 등록 가능)
2     def wrapper(obj):
3         setattr(Class, obj.__name__, obj)
4     return wrapper
```

```
1 class A:
2     def __init__(self):
3         self.b = 1
4
5 a = A()
```

```
1 @add_to_class(A) # 클래스 정의 이후에도 새로운 메서드 추가 가능
2 def do(self):    # 클래스 정의 이후 새로운 method do를 동적으로 추가
3     print('Class attribute "b" is', self.b) # self.b : 인스턴스 속성 b에 접근
4 a.do()
```

⇒ Class attribute "b" is 1

```
1 class HyperParameters: # hyperparameter 관리
2     def save_hyperparameters(self, ignore=[]):
3         raise NotImplemented
```

```
1 class B(d2l.HyperParameters):
2     """
3     B 클래스는 d2l.HyperParameters를 상속받아
4     save_hyperparameters 메서드를 사용하여 하이퍼파라미터를
5     자동으로 클래스 속성으로 저장
6     """
7     def __init__(self, a, b, c):
8         self.save_hyperparameters(ignore=['c'])
9         print('self.a =', self.a, 'self.b =', self.b)
10        print('There is no self.c =', not hasattr(self, 'c'))
11
12 b = B(a=1, b=2, c=3)
```

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

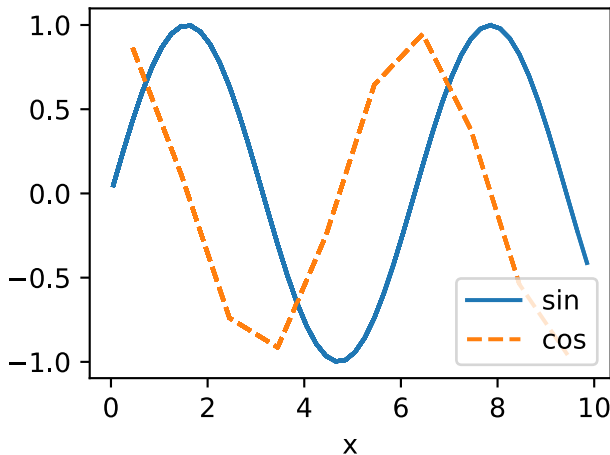
```
1 class ProgressBoard(d2l.HyperParameters):
2     def __init__(self, xlabel=None, ylabel=None, xlim=None, ylim=None, xscale='linear', yscale='linear', ls=['-', '-'],
3         self.save_hyperparameters()
```

```

4
5 def draw(self, x, y, label, every_n=1):
6     raise NotImplemented

1 board = d2l.ProgressBoard('x')
2 for x in np.arange(0, 10, 0.1):
3     board.draw(x, np.sin(x), 'sin', every_n=2) #every_n=2는 2개의 x 값마다 한 번씩 np.sin(x) 값을 그린다라는 의미
4     board.draw(x, np.cos(x), 'cos', every_n=10) #every_n=10은 10개의 x 값마다 한 번씩 np.cos(x) 값을 그린다라는 의미
5
6 # ProgressBoard 클래스를 사용하여 sin(x)와 cos(x) 함수의 값을 시각화하는 예제

```



```

1 class Module(nn.Module, d2l.HyperParameters):
2     #Module is a subclass of nn.Module (providing convenient features for handling neural network)
3     def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1): # learnable parameters 저장
4         super().__init__()
5         self.save_hyperparameters()
6         self.board = ProgressBoard()
7
8     def loss(self, y_hat, y):
9         raise NotImplementedError
10
11    def forward(self, X): # 출력 계산
12        assert hasattr(self, 'net'), 'Neural network is defined'
13        return self.net(X)
14
15    def plot(self, key, value, train): #used for visualization
16        assert hasattr(self, 'trainer'), 'Trainer is not inited' #예외처리
17        self.board.xlabel = 'epoch'
18        if train: #train data라면
19            x = self.trainer.train_batch_idx / \
20                self.trainer.num_train_batches # 현재 에포크 내에서의 진행 상태 (배치 기준)
21            n = self.trainer.num_train_batches / \
22                self.plot_train_per_epoch # 얼마나 자주 플롯을 그릴지
23        else:
24            x = self.trainer.epoch + 1 # 에포크 기준
25            n = self.trainer.num_val_batches / \
26                self.plot_valid_per_epoch #검증데이터 얼마나 자주 시각화 할지
27        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
28                        ('train_' if train else 'val_') + key,

```

```

29         every_n=int(n))
30
31     def training_step(self, batch): # training data batch를 받아, 해당 batch에서 loss value 반환
32         l = self.loss(self(*batch[:-1]), batch[-1])
33         self.plot('loss', l, train=True)
34         return l
35
36     def validation_step(self, batch): # batch에 대한 평가 지표 계산
37         l = self.loss(self(*batch[:-1]), batch[-1])
38         self.plot('loss', l, train=False)
39
40     def configure_optimizers(self): # 최적화 알고리즘 optimizer 반환
41         raise NotImplementedError

```



```

1 class DataModule(d2l.HyperParameters):
2     # data loader (data를 batch단위로 반환)생성
3     def __init__(self, root='./data', num_workers=4): # 데이터를 준비
4         self.save_hyperparameters()
5
6     def get_dataloader(self, train):
7         raise NotImplementedError
8
9     def train_dataloader(self): # 훈련 데이터셋을 위한 데이터 로더를 반환
10        return self.get_dataloader(train=True)
11
12    def val_dataloader(self): # 검증 데이터셋을 위한 데이터 로더를 반환
13        return self.get_dataloader(train=False)

```



```

1 class Trainer(d2l.HyperParameters):
2     # Trainer 클래스는 Module 클래스에서 정의된 학습 가능한 parameter를 Datamodule에서 제공되는 데이터로 훈련
3     def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
4         self.save_hyperparameters()
5         assert num_gpus == 0, 'No GPU support yet'
6
7     def prepare_data(self, data):
8         self.train_dataloader = data.train_dataloader()
9         self.val_dataloader = data.val_dataloader()
10        self.num_train_batches = len(self.train_dataloader)
11        self.num_val_batches = (len(self.val_dataloader)
12                                if self.val_dataloader is not None else 0)
13
14    def prepare_model(self, model):
15        model.trainer = self
16        model.board.xlim = [0, self.max_epochs]
17        self.model = model
18
19    def fit(self, model, data): # mddel : 학습 가능한 파라미터를 가진 모델 / data: dataset과 dataloader포함. (모델을
20        self.prepare_data(data)
21        self.prepare_model(model)
22        self.optim = model.configure_optimizers()
23        self.epoch = 0
24        self.train_batch_idx = 0
25        self.val_batch_idx = 0

```



```

26 for self.epoch in range(self.max_epochs): # 한번의 epoch는 전체 dataset을 한 번 모두 학습하는 것을 의미
27     self.fit_epoch()
28
29 def fit_epoch(self):
30     raise NotImplementedError

```

✓ 3.4. Linear Regression Implementation from Scratch

```

1 %matplotlib inline
2 import torch
3 from d2l import torch as d2l

```

```

1 # initialize weights by drawing random numbers from
2 # a normal distribution with mean 0 and a standard deviation of 0.01
3 class LinearRegressionScratch(d2l.Module):
4     def __init__(self, num_inputs, lr, sigma=0.01):
5         super().__init__()
6         self.save_hyperparameters()
7         self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
8         # weight는 mean이 0, 표준편차는 sigma, requires_grad=True: 자동 gradient 계산 가능
9         self.b = torch.zeros(1, requires_grad=True)
10        # bias는 크기가 1인 0으로 초기화.

```

```

1 @d2l.add_to_class(LinearRegressionScratch)
2 # forward method를 LinearRegressionScratch 클래스에 추가
3 def forward(self, X): # forward pass: input data X가 주어졌을때, linear regression model의 예측값 계산
4     return torch.matmul(X, self.w) + self.b
5     # y = Xw + b (X는 input data, w는 model weight)

```

```

1 @d2l.add_to_class(LinearRegressionScratch)
2 def loss(self, y_hat, y): # y_hat(예측값)과 y()사이의 오차 측정
3     l = (y_hat - y) ** 2 / 2 # 제곱 손실 함수.. 오차 계산 → parameter update
4     return l.mean()

```

```

1 class SGD(d2l.HyperParameters): # minibatch SGD
2     def __init__(self, params, lr): #lr : learning rate
3         self.save_hyperparameters()
4
5     def step(self): # parameter update method
6         for param in self.params:
7             param -= self.lr * param.grad # parameter에서 학습률*경사를 뺌. ⇒ 파라미터를 경사 방향으로 업데이트.
8             # ⇒ 손실을 줄이기 위해 경사의 반대 방향으로 파라미터 이동
9
10    def zero_grad(self): # 기울기 초기화 method (기울기가 누적되지 않게 0으로 초기화)
11        for param in self.params:
12            if param.grad is not None:
13                param.grad.zero_()
14        # 역전파가 진행될 때마다 반드시 실행되어야

```

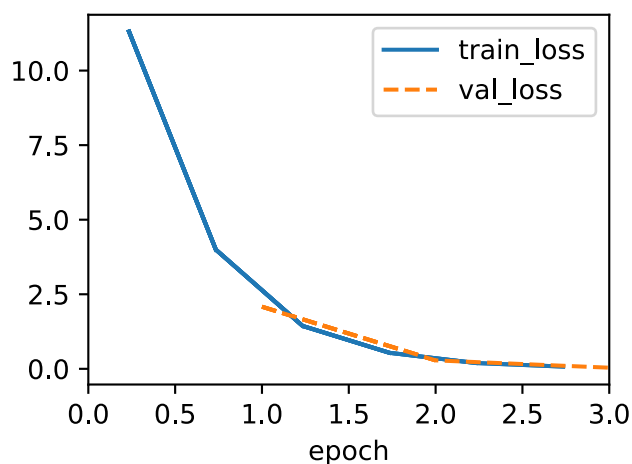
```

1 @d2l.add_to_class(LinearRegressionScratch)
2 def configure_optimizers(self): # 최적화 알고리즘을 설정하는 method
3     return SGD([self.w, self.b], self.lr)

1 @d2l.add_to_class(d2l.Trainer)
2 def prepare_batch(self, batch): # 학습 batch 처리하는 부분
3     return batch
4
5 @d2l.add_to_class(d2l.Trainer)
6 def fit_epoch(self):          # 하나의 epoch동안 모델을 학습하는 과정
7     self.model.train()        # 모델을 train 모드로 전환
8     for batch in self.train_dataloader:
9         loss = self.model.training_step(self.prepare_batch(batch)) # batch 처리 & 손실 계산하는 training_step 메서드
10        self.optim.zero_grad()    # 기울기 gradient를 모두 0으로 초기화
11        with torch.no_grad():
12            loss.backward()        # backpropagation 수행 → 각 parameter에 대한 기울기 계산
13            if self.gradient_clip_val > 0: # gradient clipping 수행 (큰 기울기 발생 → 기울기의 값을 임계값 이하로 제한하는)
14                self.clip_gradient(self.gradient_clip_val, self.model)
15            self.optim.step()      # 최적화 알고리즘의 단계 수행 → parameter 업데이트
16        self.train_batch_idx += 1
17    if self.val_dataloader is None:
18        return
19    self.model.eval()             # 모델을 검증 모드로 전환
20    for batch in self.val_dataloader: # backpropagation없이 검증 수행
21        with torch.no_grad():
22            self.model.validation_step(self.prepare_batch(batch))
23        self.val_batch_idx += 1
24
25 #number of epochs and learning rate are all hyperparameters

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

```



```

1 with torch.no_grad():
2     print(f'error in estimating w: {data.w - model.w.reshape(data.w.shape)}')
3     print(f'error in estimating b: {data.b - model.b}')

```



error in estimating w: tensor([0.1046, -0.1321])
 error in estimating b: tensor([0.2199])

✓ 4.1. Softmax Regression

4.1.1 Classification

how to represent the labels?

1) Use number labels: $y \in 1, 2, 3$, where integers represent *dog*, *cat*, *chicken* respectively

-> ordinal regression으로도 취급 가능

2) Use one-hot-encoding.

label y would be a three-dimensional vector, with $(1, 0, 0)$ as dog, $(0, 1, 0)$ as cat, and $(0, 0, 1)$ as chicken

Linear Model

- Softmax regression : linear model for multi class classification

E.g.) since we have 4 features and 3 possible output categories:

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,$$

$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,$$

$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.$$

-> just as in linear regression, we use single-layer neural network.

==> $\mathbf{0} = \mathbf{W}\mathbf{x} + \mathbf{b}$, our wights into a 3x4 matrix, and $\mathbf{b} \in \mathbb{R}^3$

The Softmax

suitable loss function을 통해 difference between o 와 y 를 minimize. but 문제가 발생:

- outputs o_i 들이 1로 합쳐지지 않을 가능성 존재
- outputs o_i 들이 negative일 수도..

=> output을 "squish"해야! (출력값을 0과 1사이의 값으로 변환해야)

해결방법:

1. Probit model:

output이 corrupted version of \mathbf{y} 인데, corruption은 normal distribution에 noise ϵ 을 더한것. ->

$$\mathbf{y} = \mathbf{0} + \epsilon$$

2. exponential function 사용:

- output을 지수 함수 형태로 변환. -> 출력값의 지수함수 변환의 합을 1로 normalization.

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

softmax operation은 ordering을 보존 -> 어떤 class가 highest probability를 가지는지 compute할 필요 X

$$\arg \max_j \hat{y}_j = \arg \max_j o_j$$

Vectorization

- we vectorize calculations in minibatches of data to **improve computational efficiency**

if) minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ of n examples with d dimensionality. q categories in the output

-> weights $\mathbf{W} \in \mathbb{R}^{d \times q}$, bias $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

=> output $\mathbf{O} = \mathbf{XW} + \mathbf{b}$.

output 확률 $\hat{Y} = \text{softmax}(\mathbf{O})$

4.1.2. Loss Function

Log-Likelihood

softmax function의 output \hat{y} : conditional probabilities of each class, given any input \mathbf{x} .

-> features \mathbf{X} and \mathbf{Y} are represented using a one-hot encoding label vector.

- 각 label $y^{(i)}$ 가 주어진 입력 $x^{(i)}$ 에 대해 독립적으로 추출

$P(\mathbf{Y} | \mathbf{X}) = \prod_{i=1}^n P(y^{(i)} | x^{(i)})$, 여기서 $P(y^{(i)} | x^{(i)})$ 는 softmax함수를 사용해 예측

- Since maximizing the product of terms is awkward -> take negative logarithm to obtain the equivalent problem of minimizing the negative log-likelihood

$-\log P(\mathbf{Y} | \mathbf{X}) = \sum_{i=1}^n -\log P(y^{(i)} | x^{(i)}) = \sum_{i=1}^n l(y^{(i)}, \hat{y}^{(i)})$, 여기서 loss function l 은 $l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^q y_j \log \hat{y}_j$. (q는 class의 수)

loss function의 특성:

- cross entropy loss function은 항상 0이상의 값. 예측이 들어맞으면 loss는 0.
- 예측확률이 0이거나 실제 값과 다름 -> loss는 매우 커짐
- 특정 class에 0의 확률을 할당 -> 만약 이게 정답이라면... loss는 커짐... $-\log 0 = \infty$ 니까...

Sortmax and Cross-Entropy Loss

$l(y, \hat{y})$ 는 cross-entropy loss function. Since softmax function and the corresponding cross-entropy loss are so common: loss의 definition을 softmax definition을 이용해 표현할 수 있음.

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

-> 손실 계산 시 모델이 각 class에 대해 예측한 확률과 실제 label이 얼마나 일치하는지

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

-> loss 함수의 derivative. derivative 는 difference between the probability (softmax operation) and what actually happened (one-hot label vector)

- Cross entropy loss는 probability vector에도 사용 가능

Entropy

- Entropy: data에 포함된 information의 양을 측정하는 척도

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

여기서 $P(j)$ 는 사건 j 가 발생할 확률. \log 는 자연로그.

-> 이 entropy는 각 사건이 발생할 확률과 그 사건에 대한 정보량을 곱한 후, 모든 사건에 대해 합산하는 방식으로 계산. -> 확률이 낮은 사건일수록 더 많은 정보량을 가지며, 반대로 확률이 높은 사건은 적은 정보량을 가짐.

we need at least $H[P]$ "nats" to encode (여기서 "nat"이란 bit와 같은데, using a code with base e)

Surprisal

- unexpected 사건이 발생 -> 얼마나 놀라운지를 측정!

$$\log \frac{1}{P(j)} = -\log P(j)$$

확률 $P(j)$ 가 작을수록 surprisal 값은 커짐. -> 드물게 발생하는 사건일 수록 더 놀라움

Cross-Entropy Revisited

cross entropy: 관측되는 data는 실제로 분포 P 에 따라 생성되었지만, 주어진 확률 분포 Q 로 예측했을 때 surprisal 기대값.

$$H(P, Q) = \sum_j -P(j) \log Q(j)$$

✓ 4.2. The Image Classification Dataset

```
1 %matplotlib inline
2 import time
3 import torch
4 import torchvision
5 from torchvision import transforms
6 from d2l import torch as d2l
7
8 d2l.use_svg_display()
```

```
1 class FashionMNIST(d2l.DataModule):
2     def __init__(self, batch_size=64, resize=(28,28)):
3         super().__init__()
```

```

4 self.save_hyperparameters()
5 trans = transforms.Compose([transforms.Resize(resize), transforms.ToTensor()]) # 이미지 전처리
6 self.train = torchvision.datasets.FashionMNIST( # train용 FashionMNIST dataset 저장
7     root=self.root, train=True, transform=trans, download=True)
8 self.val = torchvision.datasets.FashionMNIST( # validate용 FashionMNIST dataset 저장
9     root=self.root, train=False, transform=trans, download=True)

```

```

1 data = FashionMNIST(resize=(32, 32))
2 len(data.train), len(data.val)
3 #Consequently the training set and the test set contain 60,000 and 10,000 images, respectively.

```



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>
 100%|████████████████████| 26421880/26421880 [00:02<00:00, 12366034.58it/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
 100%|████████████████████| 29515/29515 [00:00<00:00, 199809.35it/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>
 100%|████████████████████| 4422102/4422102 [00:01<00:00, 3566362.81it/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
 100%|████████████████████| 5148/5148 [00:00<00:00, 16738199.22it/s]Extracting ../data/FashionMNIS

(60000, 10000)



```

1 data.train[0][0].shape
2 #tensor로 image저장할 때는, c x h x w.
3 #c: 이미지 색상 채널 수 (3채널 rgb의 경우, c=3)/ h: 이미지 높이 / w: 이미지 너비

```



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

```

1 @d2l.add_to_class(FashionMNIST)
2 def text_labels(self, indices):
3     labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
4     return [labels[int(i)] for i in indices] # 숫자 라벨을 labels 리스트에서 찾아 해당 텍스트를 반환하는 과정

```

```

1 @d2l.add_to_class(FashionMNIST)
2 def get_dataloader(self, train): # train이 True: train dataset 반환, False, val dataset 반환
3     data = self.train if train else self.val
4     return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
5                                         num_workers=self.num_workers)
6 # train / val 데이터를 minibatch 단위로 load

```

```

1 X, y = next(iter(data.train_dataloader()))
2 print(X.shape, X.dtype, y.shape, y.dtype)
3 # load a minibatch of images by invoking train_dataloader (contains 64 images)

```

⚡ /usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader has worker_init_fn=None, which is not recommended. Please use worker_init_fn=set_seed() to make this data loader deterministic. If you have a custom seed function, you can use the 'worker_init_fn' argument to disable this warning.

warnings.warn(_create_warning_msg(
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64

```

1 tic = time.time()
2 for X, y in data.train_dataloader():
3     continue
4 f'{time.time()-tic:.2f} sec'
5 # how much time it takes to read the images

```

⚡ '13.26 sec'

```

1 def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
2     raise NotImplementedError # skipping implementation detail

```

```

1 @d2l.add_to_class(FashionMNIST)
2 def visualize(self, batch, nrows=1, ncols=8, labels=[]): # batch data 시각화 nrows: 이미지 표시할 때 사용할 row
3     X, y = batch
4     if not labels: # 사용자가 label 제공 X
5         labels = self.text_labels(y) # 다시 label을 text label로 변환
6     d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels) # X.squeeze(1) 텐서의 두번째 차원 제거
7     batch = next(iter(data.val_dataloader()))
8     data.visualize(batch)

```

⚡

ankle boot	pullover	trouser	trouser	shirt	trouser
					

✓ 4.3. The Base Classification Model

```

1 import torch
2 from d2l import torch as d2l

```

```

1 class Classifier(d2l.Module):
2     def validation_step(self, batch):
3         # batch에서 수행한 예측 Y_hat과 실제 label batch[-1] 비교 → loss와 accuracy 계산
4         Y_hat = self(*batch[:-1]) # 모델에 입력 데이터를 넣어서 예측값을 얻는 부분
5         self.plot('loss', self.loss(Y_hat, batch[-1]), train=False) # 예측값과 실제값 비교 → loss 값 계산
6         self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False) # 예측값과 실제값 비교 → acc 값 계산

```

```

1 @d2l.add_to_class(d2l.Module)
2 def configure_optimizers(self):
3     return torch.optim.SGD(self.parameters(), lr=self.lr)
4     # By default we use a stochastic gradient descent optimizer, operating on minibatches

1 @d2l.add_to_class(Classifier)
2 def accuracy(self, Y_hat, Y, averaged=True):
3     Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1])) # Y_hat 차원 조정
4     preds = Y_hat.argmax(axis=1).type(Y.dtype) # 확률이 가장 높은 class 선택
5     compare = (preds == Y.reshape(-1)).type(torch.float32) # 모델이 예측한 값 preds와 실제 Y비교
6     return compare.mean() if averaged else compare    #평균 계산 → 전체 정확도

```

✓ 4.4. Softmax Regression Implementation from Scratch

```

1 import torch
2 from d2l import torch as d2l

```

```

1 X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
2 X.sum(0, keepdims=True), X.sum(1, keepdims=True)
3 # sum(0)은 axis=0을 기준으로 row를 따라 합
4 # sum(1)은 axis=1을 기준으로 column을 따라 합

```

```

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

```

```

1 # Computing softmax requires three steps
2 def softmax(X):
3     X_exp = torch.exp(X) # 1. exponentiation of each term
4     partition = X_exp.sum(1, keepdims=True)
5     # 2. sum over each row to compute the normalization constant for each example
6     return X_exp / partition #3. division of each row by its normalization constant

```

```

1 X = torch.rand((2, 5)) # 2×5 random matrix
2 X_prob = softmax(X) #softmax → probability로
3 X_prob, X_prob.sum(1) #X_prob 각 row에 대해 합 계산 → 1.

```

```

⇒ (tensor([[0.3035, 0.2504, 0.1478, 0.1291, 0.1691],
           [0.1471, 0.1902, 0.2666, 0.2164, 0.1797]]),
   tensor([1.0000, 1.0000]))

```

```

1 # raw data : consists of 28 × 28 pixel images
2 # in softmax regression, the number of outputs from network = the number of classes
3
4 class SoftmaxRegressionScratch(d2l.Classifier):
5     def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
6         super().__init__()
7         self.save_hyperparameters()

```



```

8 self.W = torch.normal(0, sigma, size=(num_inputs, num_outputs),
9                           requires_grad=True) # initialize Weights with Gaussian noise
10 self.b = torch.zeros(num_outputs, requires_grad=True) #initialize bias as zeros
11
12 def parameters(self):
13     return [self.W, self.b]

```

```

1 # define how the network maps each input to an output
2 @d2l.add_to_class(SoftmaxRegressionScratch)
3 def forward(self, X):
4     X = X.reshape((-1, self.W.shape[0]))
5     return softmax(torch.matmul(X, self.W) + self.b)

```

```

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

```

⇒ tensor([0.1000, 0.5000])

```

1 # implement the cross-entropy loss function
2 # recall that cross-entropy takes the negative log-likelihood of the predicted probability assigned to the true class
3 def cross_entropy(y_hat, y):
4     return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()
5
6 cross_entropy(y_hat, y)

```

⇒ tensor(1.4979)

```

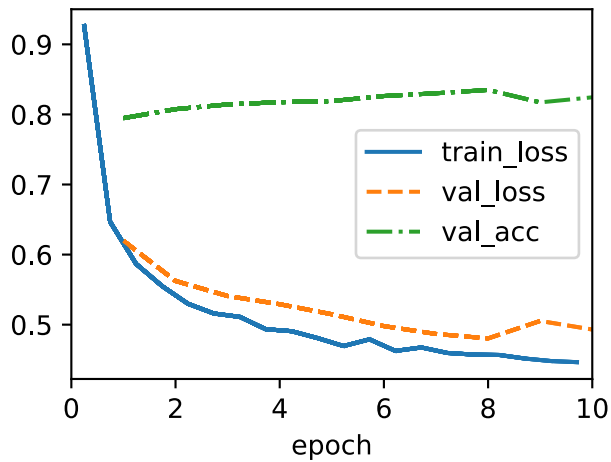
1 @d2l.add_to_class(SoftmaxRegressionScratch)
2 def loss(self, y_hat, y):
3     return cross_entropy(y_hat, y)

```

```

1 data = d2l.FashionMNIST(batch_size=256)
2 model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
3 trainer = d2l.Trainer(max_epochs=10)
4 trainer.fit(model, data)
5 # validation set을 사용해서 하이퍼파라미터를 선택하게 됨.
6 # FashionMNIST의 test data를 validation set으로 간주,, validation loss와 acc 파

```

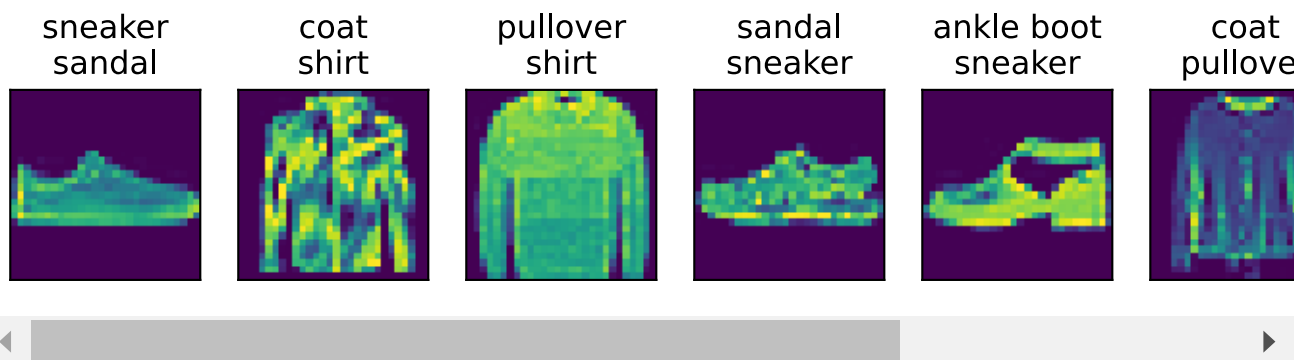


```
1 # now we are ready to classify some images
2 X, y = next(iter(data.val_dataloader()))
3 preds = model(X).argmax(axis=1)
4 preds.shape
```



```
torch.Size([256])
```

```
1 # 잘못 예측된 이미지를 시각화
2 wrong = preds.type(y.dtype) != y # 모델이 예측한 값 preds
3 X, y, preds = X[wrong], y[wrong], preds[wrong]
4 labels = [a+'\n'+b for a, b in zip(
5     data.text_labels(y), data.text_labels(preds))]
6 data.visualize([X, y], labels=labels)
7
8 # → 모델의 약점 시각적으로 확인 가능.
```



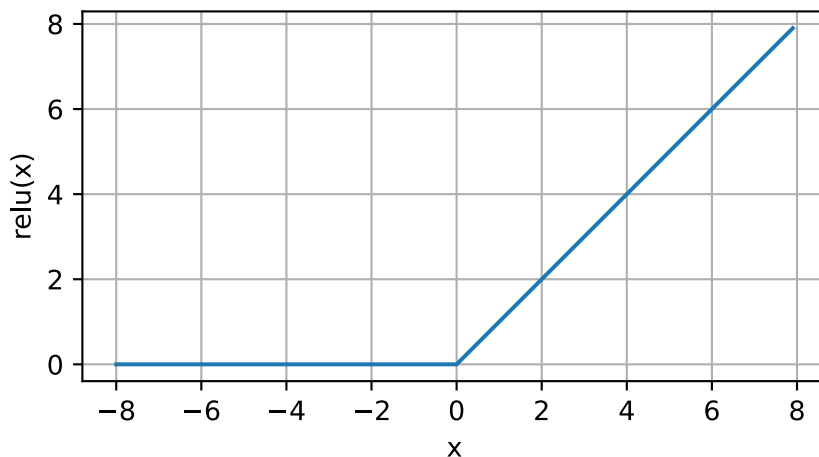
✓ 5.1. Multilayer Perceptrons

```
1 %matplotlib inline
2 import torch
3 from d2l import torch as d2l
```

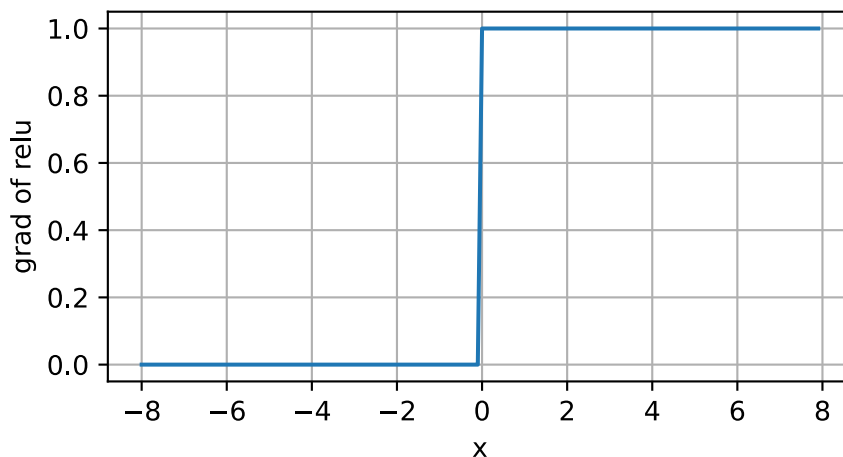
$$\text{ReLU}(x) = \max(x, 0)$$

the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0

```
1 x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
2 y = torch.relu(x)
3 d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
4
5 # the input is negative, the derivative of the ReLU function is 0
6 # when the input is positive, the derivative of the ReLU function is 1
7 # the ReLU function is not differentiable when the input takes value precisely equal to 0
```



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



reason for using ReLU: its derivatives are particularly well behaved -> optimization better behaved & vanishing gradient problem 완화

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

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

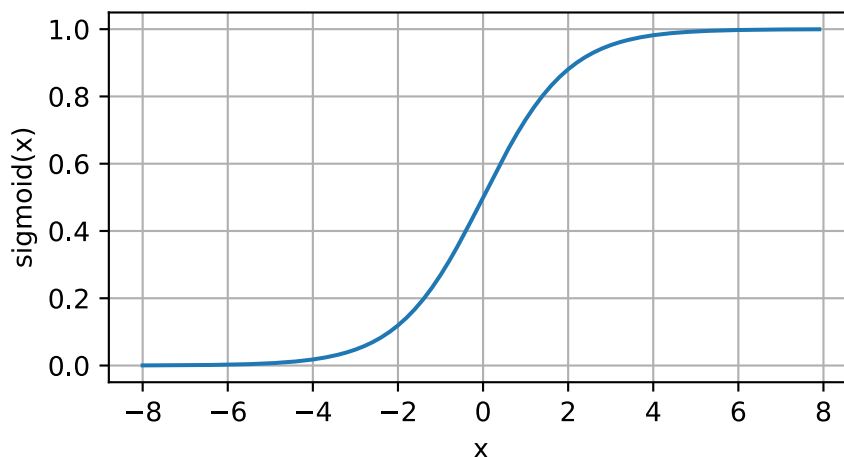
it is a smooth, differentiable approximation to a thresholding unit. However, the sigmoid has largely been replaced by the simpler and more easily trainable ReLU for most use in hidden layers.

->reason: the sigmoid poses challenges for optimization since its gradient vanishes for large positive and negative arguments. 양의 큰 값이나 음의 큰 값을 입력으로 할 때 시그모이드는 plateaus를 형성해 학습이 정체될 수 있음

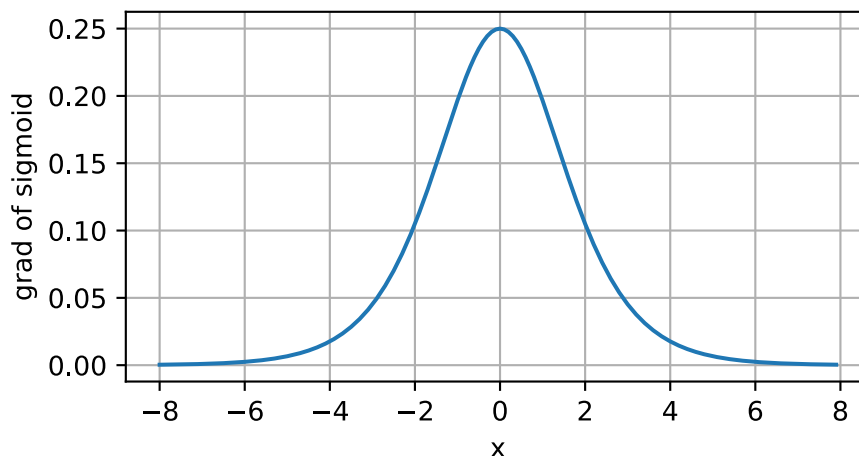
the derivative of sigmoid function:

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$$

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



```
1 # derivative of the sigmoid function
2 if x.grad is not None:
3     x.grad.zero_() # 이전에 계산된 gradient를 초기화
4 y.backward(torch.ones_like(x), retain_graph=True)
5 d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



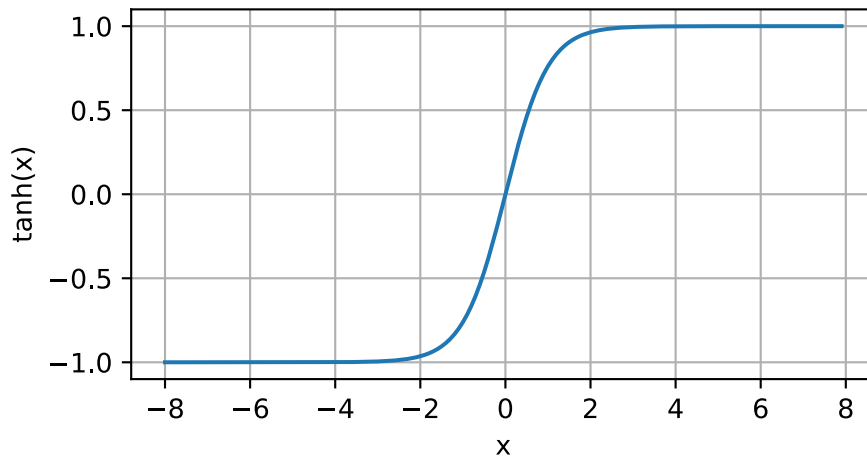
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

tanh function also squashes its inputs like sigmoid function (but tanh: from -1 to 1)

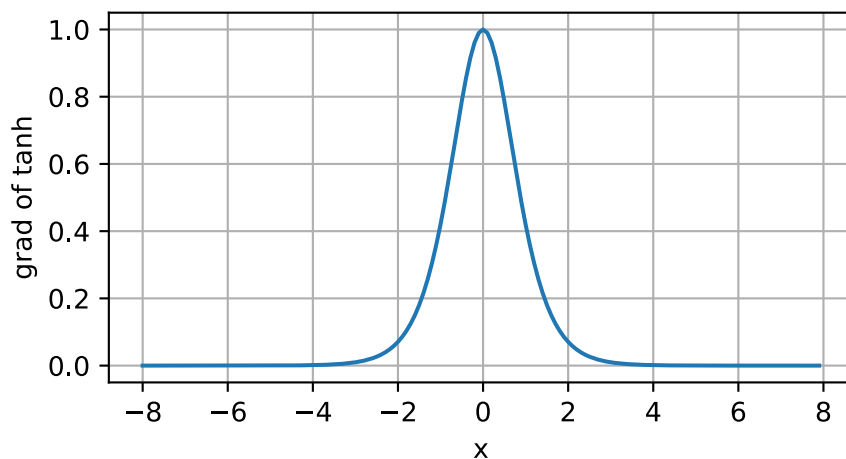
- the derivative of tanh function:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

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



```
1 # derivative of the tanh function
2 if x.grad is not None:
3     x.grad.zero_() # 이전에 계산된 gradient를 초기화
4 y.backward(torch.ones_like(x), retain_graph=True)
5 d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



✓ 5.2. Implementation of Multilayer Perceptrons

```
1 import torch
2 from torch import nn
3 from d2l import torch as d2l
```

```

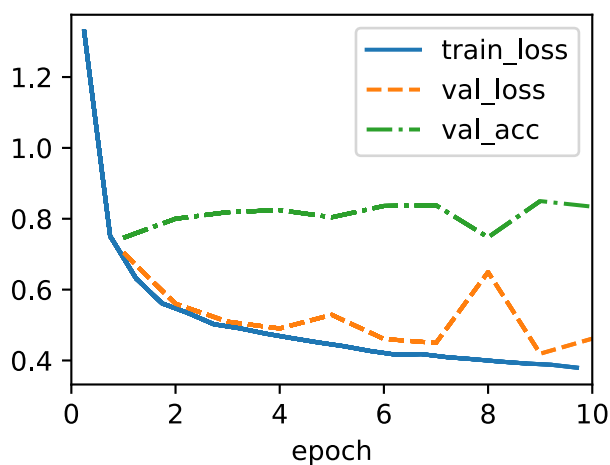
1 # initializing model parameters
2
3 class MLPScratch(d2l.Classifier):
4     def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
5         super().__init__()
6         self.save_hyperparameters()
7         # parameter 초기화
8         self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
9         self.b1 = nn.Parameter(torch.zeros(num_hiddens))
10        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
11        self.b2 = nn.Parameter(torch.zeros(num_outputs))

1 # model
2
3 def relu(X): # ReLU function
4     a = torch.zeros_like(X)
5     return torch.max(X, a)

1 @d2l.add_to_class(MLPScratch)
2 # input data가 주어졌을 때 MLP 모델을 통해 forward pass 수행
3 def forward(self, X):
4     X = X.reshape((-1, self.num_inputs))
5     # reshape each two-dimensional image into a flat vector of length num_inputs.
6     H = relu(torch.matmul(X, self.W1) + self.b1) # 은닉층 출력 계
7     return torch.matmul(H, self.W2) + self.b2 # 출력층의 값을 계산

1 # training
2
3 model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
4 data = d2l.FashionMNIST(batch_size=256)
5 trainer = d2l.Trainer(max_epochs=10)
6 trainer.fit(model, data)

```



```

1 # Concise Implementation Model
2
3 class MLP(d2l.Classifier):
4     def __init__(self, num_outputs, num_hiddens, lr):
5         super().__init__()

```

```

6 self.save_hyperparameters()
7 self.net = nn.Sequential(nn.Flatten(), nn.Linear(num_hiddens), nn.ReLU(), nn.Linear(num_output

1 """
2 MLP inherits the forward method from the Module class (Section 3.2.2) to simply invoke self.net(X) (X is inp
3 which is now defined as a sequence of transformations via the Sequential class.
4 """

```

```

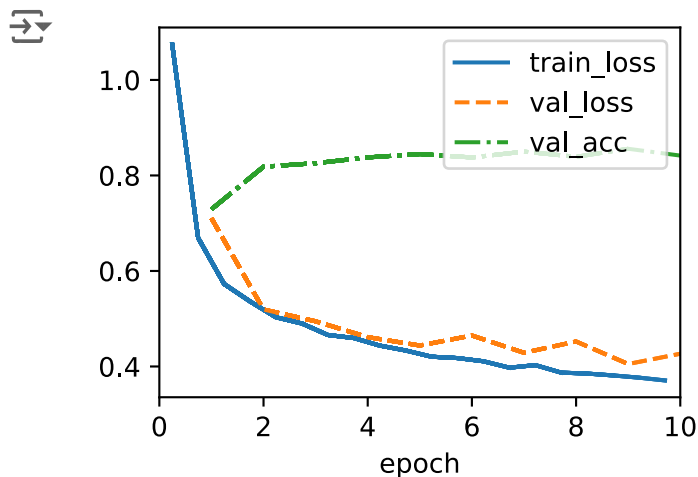
⇒ '\nMLP inherits the forward method from the Module class (Section 3.2.2) to simply invoke self.net(X) (X
is input),\nwhich is now defined as a sequence of transformations via the Sequential class.\n'

```

```

1 # Concise Implementation Training
2
3 model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
4 trainer.fit(model, data)

```



5.3. Forward Propagation, Backward Propagation, and Computational Graphs

Forward Propagation

- Forward propagation: the calculation and storage of intermediate variables (including outputs) for a neural network in order from the input layer to the output layer.

Assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and hidden layer doesn't include bias.

- intermediate variable \mathbf{z} :

$\mathbf{z} = \mathbf{W}^{(1)} \mathbf{x}$, 여기서 \mathbf{x} 는 input example, $\mathbf{W}^{(1)}$ 는 weight parameter of hidden layer

- hidden layer output \mathbf{h} :

$\mathbf{h} = \phi(\mathbf{z})$, ϕ 는 activation function. 즉, intermediate variable에 activation function을 적용해, hidden layer output을 계산.

- output layer \mathbf{o} :

$\mathbf{o} = \mathbf{W}^{(2)} \mathbf{h}$, assuming that the parameters of the output layer possess only a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$.

- loss function L :

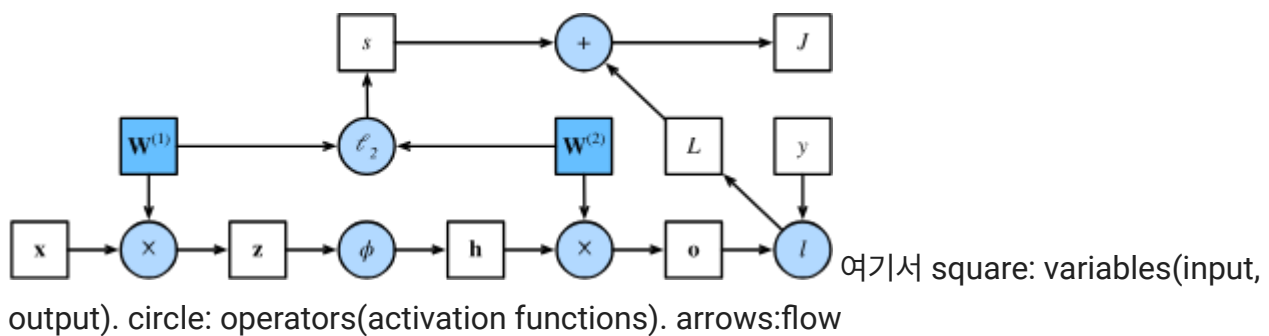
$L = l(\mathbf{o}, y)$, \mathbf{o} is estimated result, and y is the real label

- l_2 regularization s :

$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right)$, giving penalties so that \mathbf{W} doesn't get too big. Here, the Frobenius norm of the matrix is simply the l_2 norm applied after flattening the matrix into a vector.

- Final model's regularized loss on a given data example: $J = L + s$

Computational Graph of Forward Propagation



Backpropagation

- the method of calculating the gradient of neural network parameters. 여기서 chain rule을 이용해 계산.

Assume:

we have functions $Y = f(X)$ and $Z = g(Y)$. input X , output Y and Z 는 임의의 형태를 가진 tensor. chain rule을 써서, X 에 대한 Z 의 derivative 구할 수 있음.

$\frac{\partial Z}{\partial X} = \text{prod} \left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right)$, prod 연산자는 두 미준 값을 곱해 최종 결과를 도출.

Backpropagation을 사용해 objective function J 에 대한 가중치 행렬 $\mathbf{W}^{(1)}$ 과 $\mathbf{W}^{(2)}$ 의 gradient를 계산하는 과정:

- objective function $J = L + s$ 를 loss term L 과 regularization term s 에 대한 gradient를 계산.

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

- the objective function을 output layer \mathbf{o} 의 variable에 대한 gradient를 계산. (chain rule 사용)

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left(\frac{\partial L}{\partial J}, \frac{\partial \mathbf{o}}{\partial L} \right) = \frac{\partial \mathbf{o}}{\partial L}$$

- regularization term s 의 gradient계산.

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

4. output layer에 가장 가까운 model parameter $\mathbf{W}^{(2)}$ 의 gradient 계산.

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

5. hidden layer output \mathbf{h} 의 gradient 계산.

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

6. hidden layer output인 h 는 activation function ϕ 에 의해 계산되니, intermediate variable z 의 gradient 계산할 때는 elementwise multiplication operator 필요.

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

7. 드디어, input layer에 가장 가까운 model parameter $\mathbf{W}^{(1)}$ 의 gradient 계산.

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

Training Neural Networks

- forward and backward propagation depend on each other

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.

Therefore when training neural networks, once model parameters are initialized, we alternate forward propagation with backpropagation, updating model parameters using gradients given by backpropagation. (이때, backpropagation은 stored intermediate values from forward propagation을 reuse 해서 중복 계산을 피함. 이것은 training이 significantly more memory를 plain prediction보다 요구하는 이유!)

✓ Discussions & Exercises

2.2.2. Data Preparation Memo

- in Supervised learning, input values와 target values로 model을 train
- data set에서 input과 target을 분리
- NaN값 (missing values)를 처리하려면: imputation (NaN을 추정치로 채움) / deletion (강 제거해버림)
- categorical input fields: NaN을 category로 treat.

2.3.1. Scalars Memo

- c and f : 일반적으로 unknown scalars
- $c = \frac{5}{9}(f - 32)$ 에서 5, 9, 32 -> constant scalar
- $x \in \mathbb{R}$: x is a real-valued scalar
- $x, y \in 0, 1$: x and y are variables that can only take values 0 or 1

2.3.2. Vectors Memo

- Vector: fixed-length array of scalars
- elements: these scalars in vector (also as entries and components)
- Vector: implemented as 1st order tensors
- denote vectors by bold lowercase


2.3.3. Matrices Memo

- scalar: 0th-order tensor // vector: 1st-order tensor // **matrix**: 2nd-order tensor
- $A \in \mathbb{R}^{m \times n}$ -> A는 m개의 row와 n개의 column을 가지는 real-valued scalar로 구성
- $a_{\{ij\}}$ -> ith row and jth column

2.3.4. Tensors Memo

- tensors: nth-order arrays로 확장 가능
- tensors: image data에 사용됨 -> 3rd order tensor (height, width, channel)
- 여기서 각 each spatial location마다 (red, green, blue)
- 이미지 여러개 -> 4차원 텐서

2.3.8. Dot Products Memo

- $x, y \in \mathbb{R}^d$ 의 dot product $x^T y$ = inner product $\langle x, y \rangle$
-  두 벡터의 대응 요소 곱한 뒤 그 곱들을 더한 것
- $x^T y = \sum_{i=1}^d x_i y_i$

if) x 가 $[0., 1., 2.]$ // y 는 $[1., 1., 1.]$ -> dot product는 $(0 \times 1 + 1 \times 1 + 2 \times 1) = 3$

2.3.9. Matrix-Vector products Memo

$$\bullet A = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix}, \text{ 여기서 } \mathbf{a}_i^T \text{ 는 행렬 } A \text{ 의 } i \text{ 번째 row vector}$$

$$-Ax = \begin{bmatrix} \mathbf{a}_1^T x \\ \mathbf{a}_2^T x \\ \vdots \\ \mathbf{a}_m^T x \end{bmatrix}$$

- \Rightarrow multiplication with a matrix $A \in \mathbb{R}^{m \times n}$ 은 transformation that projects vectors from \mathbb{R}^n to \mathbb{R}^m (벡터를 $A \in \mathbb{R}^n$ 공간에서 $A \in \mathbb{R}^m$ 공간으로 projection)
- Transpose는 matrix의 row와 column 뒤바꾸는 연산

2.3.10. Matrix-Matrix Multiplication Memo

- $A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{k \times m}$
- $C = AB, C \in \mathbb{R}^{n \times m}$
- $c_{ij} = a_i^T b_j \rightarrow A$ 의 각 row와 B 의 각 column을 내적 $\Rightarrow C$ 의 요소 계산

2.3.11. Norms Memo

- norm: tells us how big a vector is
- l_2 norm measures the length of a vector

$\|\cdot\|$ satisfies three properties:

- $\|\alpha \mathbf{x}\| = |\alpha| \|\mathbf{x}\|, \alpha \in \mathbb{R}$
- $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$
- $\|\mathbf{x}\| > 0$ for all $\mathbf{x} \neq 0$

-
- $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2} \leftarrow l_2 \text{ norm}$
 - $\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i| \leftarrow l_1 \text{ norm}$
 - $\|\mathbf{x}\|_p = (\sum_{i=1}^n |x_i|^p)^{1/p} \leftarrow l_p \text{ norm}$
-

Frobenius norm: behaves as if it were an l_2 norm of a matrix-shaped vector

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

2.4.1. Derivatives and Differentiation Memo

(differentiation: 미분, derivatives: 도함수)

- derivate: rate of change in a function with respect to changes (함수의 입력 값이 아주 조금 변했을 때, 출력 값이 얼마나 변하는지에 대한 변화율(rate of change)을 측정)
- $f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$ when $f'(x)$ exists, f is said to be differentiable at x ; when $f'(x)$ exists for all s on a set; $\rightarrow f$ is differentiable on this set.
- 딥러닝에서 loss function을 optimize 하기 위해서는 derivatives 계산하는 것이 중요 \rightarrow optimize a differentiable surrogate(대체함수) instead

$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = D_x f(x), \frac{d}{dx}$
and D are differentiation operators.

$$\frac{d}{dx} C = 0 \quad \text{for any constant } C$$

$$\frac{d}{dx} x^n = nx^{n-1} \quad \text{for } n \neq 0$$

$$\frac{d}{dx} e^x = e^x$$

$$\frac{d}{dx} \ln x = x^{-1}$$

$$\frac{d}{dx} [Cf(x)] = C \frac{d}{dx} f(x) \quad \text{Constant multiple rule}$$

$$\frac{d}{dx} [f(x) + g(x)] = \frac{d}{dx} f(x) + \frac{d}{dx} g(x) \quad \text{Sum rule}$$

$$\frac{d}{dx} [f(x)g(x)] = f(x) \frac{d}{dx} g(x) + g(x) \frac{d}{dx} f(x) \quad \text{Product rule}$$

$$\frac{d}{dx} \left(\frac{f(x)}{g(x)} \right) = \frac{g(x) \frac{d}{dx} f(x) - f(x) \frac{d}{dx} g(x)}{g^2(x)} \quad \text{Quotient rule}$$

2.4.3. Partial Derivatives and Gradients Memo

- $y = f(x_1, x_2, \dots, x_n)$. 여기서 i 번째 변수 x_i 에 대한 partial derivative (부분미분)은:

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i+h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}$$

\rightarrow 다른 변수는 고정, x_i 를 변화시키며 변화율 구하기

- $\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = \partial_i f = D_i f = D_{x_i} f$

다 같은 의미 가짐

- gradient: 모든 변수에 대한 부분 미분을 벡터로 나타낸 것

if) 함수 $f: \mathbb{R}^n \rightarrow \mathbb{R}$ 의 input이 n -dimensional vector $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ 이고, output이 scalar라면

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^T \text{가 gradient.}$$

some following rules

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$ we have $\nabla_{\mathbf{x}} \mathbf{A} \mathbf{x} = \mathbf{A}^T$ and $\nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{A} = \mathbf{A}$.
- For square matrices (정사각행렬) $\mathbf{A} \in \mathbb{R}^{n \times n}$ we have that $\nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^T) \mathbf{x}$ and in particular (대칭행렬이라면) $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^T \mathbf{x} = 2\mathbf{x}$.
- Similarly, for any matrix \mathbf{X} , we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$.

2.4.4. Chain Rule Memo

단일 변수의 경우에, $y = f(g(x))$ 에 대해 chain rule을 적용하면:

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

여기서 $y = f(u)$, $u = g(x)$.

다변수의 경우에, 즉 y 가 여러변수 u_1, u_2, \dots, u_m 를 가지는 함수 & 각 u_i 는 x_1, x_2, \dots, x_n 의 함수일 경우에:

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1} \cdot \frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2} \cdot \frac{\partial u_2}{\partial x_i} + \dots + \frac{\partial y}{\partial u_m} \cdot \frac{\partial u_m}{\partial x_i} \text{ 이고, 이걸 } \nabla_{\mathbf{x}} y = \mathbf{A} \nabla_{\mathbf{u}} y.$$

-> gradient계산은 vector-matrix 곱셈으로 처리 가능.

2.5.2. Backward for Non-Scalar Variables Memo

- 벡터 함수의 미분 결과: 자코비안이라는 행렬로 나타남..but

딥러닝에서는 Jacobian 대신 sum up the gradients of each component y with respect to the full vector

- non-scalar tensors (vector or matrix..)에 대해 backward() 함수 호출 -> 오류 발생 가능

=> we need to provide vector \mathbf{v} such that backward will compute $\mathbf{v}^T \partial_{\mathbf{x}} y$ rather than $\partial_{\mathbf{x}} y$

2.5.3. Detaching Computation Memo

- detach(): 특정 계산이 gradient 흐름에 영향을 미치지 않도록 계산 그래프에서 분리하는 역할
- 딥러닝에서는 계산 그래프를 이용해 변수간 연산 기록, 그 그래프를 통해 역전하 과정에서 그래디언트 계산
- but 때때로, 일부 중간 계산 값이 최종 그래디언트 계산에 영향을 미치지 않도록 하려면: detach() 이 필요.

2.5.1. Basics Memo

Linear Regression

- n : the number of examples
- $x^{(i)}$ 는 i 번째 sample, $x_j^{(i)}$ 는 그 sample의 j 번째 좌표

linear regression model: features는 weighted sum으로 target을 설명할 수 있음

- weight: 각 feature가 target에 미치는 영향
- bias: make our model's predictions fit true

$\hat{y} = w_1x_1 + \dots + w_dx_d + b$ (모든 feature를 하나의 vector \mathbf{x} 에, weight를 하나의 벡터 \mathbf{w} 로 표현)

$$\Rightarrow \hat{y} = \mathbf{w}^\top \mathbf{x} + b$$

design matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ 은 n 개의 sample과 d 개의 feature. 이 matrix에서, 각 row는 하나의 sample, 각 column은 하나의 feature

$\hat{y} = \mathbf{X}\mathbf{w} + b \rightarrow \mathbf{X}$ 에 있는 모든 sample의 예측값을 한번에 계산.

여기서 bias 를 더하는 건 broadcasting으로, b 가 모든 sample에 동일하게 적용

- Linear regression의 목표는 weight vector \mathbf{w} 와 bias b 를 찾아 measurement error 줄이기

but 현실 세계에는 noise가 존재 \rightarrow noise를 포함하여 오차를 처리해야

Loss Function

- Loss Function: 실제값과 예측값 간의 거리를 정량화

완벽한 예측 \rightarrow 손실값 0

Regression problems에서 가장 흔한 loss function은 squared error

Loss Function

$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)} \right)^2 = \frac{1}{2} \rightarrow i$ 번째 sample에서 loss 계산. ($1/2$ 는 미분시 계산이 편해지는 constant)

Average Loss

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2$$

모델이 전체적으로 얼마나 잘 예측하는지

Optimization Objective

$$\mathbf{w}^*, b^* = \arg \min_{\mathbf{w}, b} L(\mathbf{w}, b)$$

\rightarrow 가중치 \mathbf{w} 와 바이어스 b 를 찾아 average loss를 최소화하는 것이 목표

** Analytic Solution **

- Linear regression은 loss function을 해석적으로 해결 가능... \rightarrow 공식으로 optimal parameters 찾을 수 있음!

1. subsume the bias b into the parameter \mathbf{w} by appending a column to the design matrix consisting of all 1s.

2. 이렇게 된다면 prediction 문제는 $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ 를 minimize하는 것이 됨. (\mathbf{y} 는 실제값 벡터, \mathbf{X} 는 feature matrix, \mathbf{w} 는 weight vector)

3. loss function minimize하기 위해 loss를 derivate with respect to \mathbf{w}

$$\rightarrow \frac{\partial}{\partial \mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \quad \text{and hence} \quad \mathbf{X}^\top \mathbf{y} = \mathbf{X}^\top \mathbf{X}\mathbf{w}$$

위의 식을 풀면, 최적의 가중치 \mathbf{w}^* 은:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

여기서 $\mathbf{X}^\top \mathbf{X}$ 가 역행렬이 존재할 때 해가 유일함.

Minibatch Stochastic Gradient Descent

Gradient Descent:

- Loss function을 minimize하기 위해 weight를 반복적으로 업데이트 하는 방법
- Loss function의 gradient를 계산 -> loss값을 줄이는 방향으로 weight 갱신 -> loss function 점점 감소

Batch Gradient Descent:

- 전체 dataset에 대해 loss function 기울기 구한 후 weight 업데이트
- 대규모 dataset 다룰 때 느릴 수 있음

Stochastic Gradient Descent:

- single sample에 대해 기울기 계산 -> weight 업데이트
- 계산은 빠르지만 data의 noise로 인해 진동이 심해질 수 있음

Mini-batch Gradient Descent:

- minibatch라는 소규모 sample 묶음을 사용 -> 기울기 계산 & 매개변수 update

minibatch SGD 업데이트 과정: 매 iteration t 마다, 크기가 $|\mathcal{B}_t|$ 인 미니배치를 랜덤하게 선택. 이후 가중치 업데이트

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \nabla_{\mathbf{w}, b} l^{(i)}(\mathbf{w}, b)$$

여기서 \mathbf{w} 와 b 는 각 weight and bias. η 는 learning rate. l^i 는 i 번째 sample에 대한 loss

$$\text{가중치 업데이트: } \mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} (\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}) \mathbf{x}^{(i)}$$

$$\text{bias 업데이트: } b \leftarrow b - \frac{\eta}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} (\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)})$$

종료 기준이 충족되면, 최종 estimated model parameters $\hat{\mathbf{w}}, \hat{b}$ 저장.

- we typically do not care about finding an exact set of parameters but merely any set of parameters that leads to accurate predictions (and thus low loss).
- The more formidable task is to find parameters that lead to accurate predictions on previously unseen data, a challenge called generalization

Predictions

$\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$ 를 사용해 새로운 example에 대한 예측값 계산 가능. -> inference 라는 단어 사용하면,, 혼란이 일어날 수 있음,, prediction이라는 말을 쓰시다!

3.1.3. The Normal Distribution and Squared Loss Memo

- optimal parameters return the conditional expectation $E[Y|X]$. 이걸 underlying pattern이 truly linear일때 적용.
- outlier에 large penalties를 적용함
- squared loss에 probabilistic assumptions about the distribution of noise 도입.
- Gaussian distribution 과 linear regression with squared loss는 deeper connection을 공유!

Normal (Gaussian) distribution: $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$

x 는 확률변수, μ 는 평균, σ^2 는 분산

- Noise가 있는 관측값의 가정:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon$$

여기서 noise ϵ 은 normal distribution $\mathcal{N}(0, \sigma^2)$ 를 따름

- Likelihood function: 주어진 data에서 특정 y 값을 얻을 확률을 나타내는 함수

$$P(y | \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right)$$

-> y 와 linear model $\mathbf{w}^\top \mathbf{x}$ 의 차이를 설명

- the Principle of Maximum Likelihood:

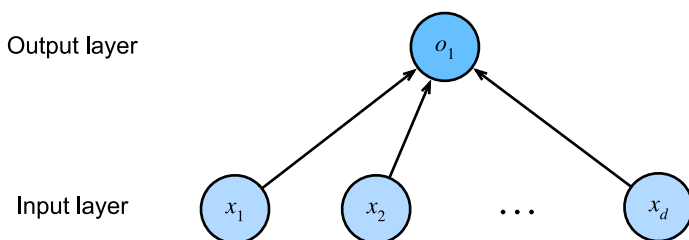
$$P(\mathbf{y} | \mathbf{X}) = \prod_{i=1}^n P(y^{(i)} | \mathbf{x}^{(i)})$$

best values of parameters \mathbf{w} and b : maximizes the likelihood of the entire dataset

- Log-likelihood function: maximizing likelihood는 어려움 -> log likelihood function을 사용해 계산을 간소화. Minimize the negative log-likelihood.

$$-\log P(\mathbf{y} | \mathbf{X}) = \sum_{i=1}^n \left(\frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b)^2 \right)$$

3.1.4. Linear Regression as a Neural Network Memo



- 입력 계층에는 feature x_1, x_2, \dots, x_d 가 있고, 이걸 하나의 출력 뉴런 o_1 에 연결 => linear regression을 single-layer로 구성된 신경망으로 볼 수 있음!

3.4.4. Training Memo

- epoch: iterating through the entire training dataset.
- In each epoch, we iterate through the entire training dataset, passing once through every example.
- In each iteration, we grab a minibatch of training examples, and compute its loss through the model's `training_step` method. (각 iteration에서: minibatch로부터 데이터를 가져와 모델의 loss 계산)
- optimization algorithm:

1) initialize parameters \mathbf{w}, \mathbf{b}

2) Repeat until done:

- Compute gradient $\mathbf{g} \leftarrow \nabla_{(\mathbf{w}, \mathbf{b})} \frac{1}{|B|} \sum_{i \in B} \ell(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, \mathbf{b})$
- Update parameters $(\mathbf{w}, \mathbf{b}) \leftarrow (\mathbf{w}, \mathbf{b}) - \eta \mathbf{g}$

5.1.1. Hidden Layers

Limitations of Linear Models

- linearity implies the weaker assumption of monotonicity. monotonicity는 feature의 increase가 항상 model's output의 증가를 유발한다고 가정.

But) while monotonic, this relationship is not linearly associated -> 해결하는 한가지 방법은 logistic map (로그변환) 사용

- monotonicity를 위반하는 example 존재. (higher body temperature만이 greater risk는 아님! 37°C아래로 떨어지면 lower temperatures가 greater risk! -> 해결 방법은,, using distance from 37°C as a feature)
- what about classifying images of cats and dogs? inverting an image를 해도... category가 preserve됨 -> simple preprocessing으로 해결 불가. (복잡한 데이터에서는 각 픽셀의 중요성이 주변 맥락에 따라 달라지기 때문)

With deep neural networks, we used observational data to jointly learn both a representation via hidden layers and a linear predictor that acts upon that representation

Incorporating Hidden Layers

- MLP: overcome the limitations of linear models by incorporating one or more hidden layers. The easiest way to do this is to stack many fully connected layers on top of one another.

MLP에서 layers는 fully connected. Every input influences every neuron in the hidden layer, and each of these in turn influences every neuron in the output layer.

From Linear to Nonlinear

- input data $\mathbf{X} \in \mathbb{R}^{n \times d}$: a minibatch of n examples where each example has d inputs
- one-hidden-layer MLP에서: $\mathbf{H} \in \mathbb{R}^{n \times h}$ 가 outputs of the hidden layer.
- hidden-layer weights: $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$, biases: $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$, output-layer weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and output-layer biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$.

So,

$\mathbf{H} = \mathbf{XW}^{(1)} + \mathbf{b}^{(1)}$, 여기서 \mathbf{O} 는 outputs of the one-hidden-layer MLP

$\mathbf{O} = \mathbf{HW}^{(2)} + \mathbf{b}^{(2)}$

- after adding the hidden layer, our model now requires us to track and update additional sets of parameter -> 이 모델에서는 새로운 것을 얻지 못함. The hidden units above are given by an affine function of the inputs, and the outputs (pre-softmax) are just an affine function of the hidden units. 아핀 함수의 아핀 함수는 여전히 아핀 함수이므로, 은닉층을 추가해도 본질적으로 선형 모델과 차이가 없음

-> Activation Function 이 필요함. (nonlinear함)

- a popular choice is the ReLU activation function, $\sigma(x) = \max(0, x)$

with activation function in place,

$\mathbf{H} = \sigma(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)})$,

$\mathbf{O} = \mathbf{HW}^{(2)} + \mathbf{b}^{(2)}$

after computing the linear portion of the layer, we can calculate each activation without looking at the values taken by the other hidden units.

++

$\mathbf{H}^{(1)} = \sigma_1(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)})$,

$\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$

이렇게 층을 쌓아가며 더 복잡한 표현을 학습하는 MLP를 구축할 수 있음

✓ 5.1.2. Activation Functions

reason for using ReLU: its derivatives are particularly well behaved -> optimization better behaved & vanishing gradient problem 완화

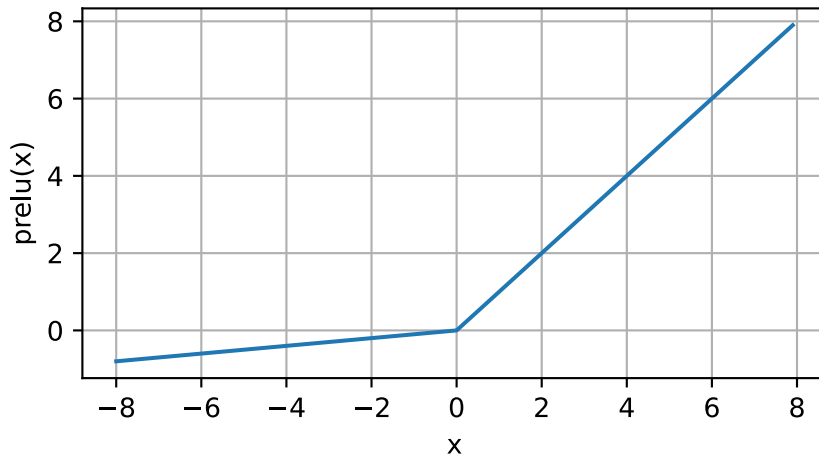
ReLU의 variants: $\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x)$

```
1 %matplotlib inline
2 import torch
3 from d2l import torch as d2l
```

```

1 def pReLU(x, alpha):
2     return torch.max(torch.zeros_like(x), x) + alpha * torch.min(torch.zeros_like(x), x)
3
1 #pReLU
2
3 alpha = 0.1
4 x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
5 y = pReLU(x, alpha)
6 d2l.plot(x.detach(), y.detach(), 'x', 'prelu(x)', figsize=(5, 2.5))

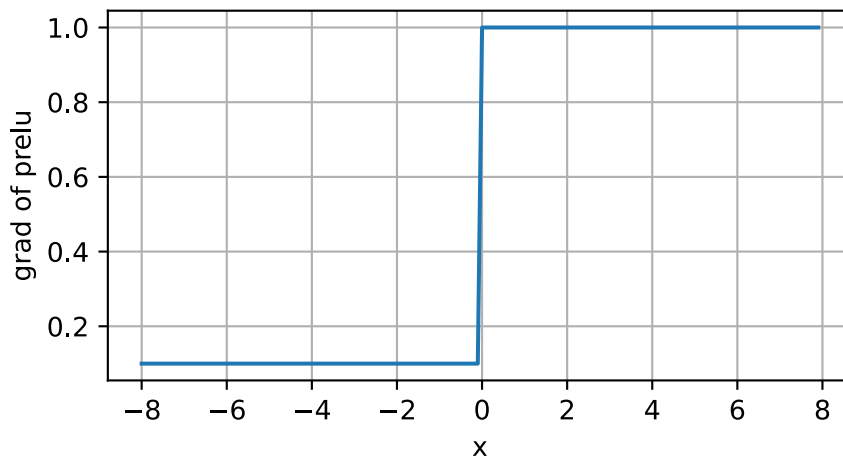
```



```

1 # derivative of the pReLU function
2 if x.grad is not None:
3     x.grad.zero_() # 이전에 계산된 gradient를 초기화
4 y.backward(torch.ones_like(x), retain_graph=True)
5 d2l.plot(x.detach(), x.grad, 'x', 'grad of prelu', figsize=(5, 2.5))

```



reCAPTCHA 서비스에 연결할 수 없습니다. 인터넷 연결을 확인한 후 페이지를 새로고침하여 reCAPTCHA 보안문자를 다시 로드하세요.