딥러닝 HW 1

2020320131

컴퓨터학과 박수빈

2.1. Data Manipulation

[1]:
```
import torch
x = torch.arange(12, dtype=torch.float32)
x
```

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

[2]:
```
x.numel()
```

[2]:
```
12
```

[3]:
```
x.shape
```

[3]:
```
torch.Size([12])
```

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

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

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

[5]:
```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

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

[6]:
```
torch.ones((2, 3, 4))
```

[6]:
```
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.]]])
```

[7]:
```
torch.rand(3, 4)
```

[7]:
```
tensor([[0.3240, 0.4352, 0.4766, 0.6256],
        [0.9865, 0.5347, 0.0884, 0.1191],
        [0.3814, 0.9570, 0.7616, 0.8191]])
```

[8]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

[8]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

[9]:
```
X[-1], X[1:3]
```

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

```
[10]: X[1, 2] = 17
      X

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

[11]: X[:2, :] = 12
      X

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

[12]: torch.exp(x)

[12]: 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])

[13]: 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

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

x의 exp 값을 봤을 때, X를 통해 값을 변경하면 x에서도 동일한 위치의 값이 함께 변경된다는 것을 알 수 있다. 이를 통해 torch.reshape()는 새로운 메모리를 할당하는 것이 아니며, 파이썬에서의 '=' 기호는 shallow copy에 해당한다는 것을 추측할 수 있다. 또한 뒤에서 언급되는 x.numpy()도 기존의 텐서와 변환된 배열이 같은 메모리 공간을 공유한다고 한다.

```
[14]: 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)

[14]: (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.]]))

[15]: X == Y, X < Y, X > Y

[15]: (tensor([[False,  True, False,  True],
               [False, False, False, False],
               [False, False, False, False]]),
       tensor([[ True, False,  True, False],
               [False, False, False, False],
               [False, False, False, False]]),
       tensor([[False, False, False, False],
               [ True,  True,  True,  True],
               [ True,  True,  True,  True]]))

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

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

```
[17]: a + b
```

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

```
[18]: before = id(Y)
      Y = Y + X
      id(Y) == before
```

```
[18]: False
```

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

```
id(Z): 2501960911664
id(Z): 2501960911664
```

```
[20]: before = id(X)
      X += Y
      id(X) == before
```

```
[20]: True
```

또한 'X=X+Y'처럼 변수를 새로 선언하는 것은 변수가 가리키는 주소값이 변하게 하지만, 'X[:]'처럼 slice notation을 이용하거나 식을 'X += Y'처럼 작성하는 것은 주소값이 변하지 않아 메모리 낭비를 막을 수 있다.

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

```
[21]: (numpy.ndarray, torch.Tensor)
```

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

```
[22]: (tensor([3.5000]), 3.5, 3.5, 3)
```

```
[23]: a = torch.arange(4).reshape((4, 1))
      b = torch.arange(3).reshape((1, 3))
      a, b, a + b, a - b
```

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

2.2. Data Preprocessing

PYTORCH MXNET JAX TENSORFLOW

```
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
```

```
import pandas as pd

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

```
  NumRooms RoofType   Price
0       NA      NaN  127500
1        2      NaN  106000
2        4    Slate  178100
3       NA      NaN  140000
```

NumRooms의 형식이 예시와 다르게 문자열로 출력되어서 to_numeric을 통해 숫자로 변환했다.

[1]:
```
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)
data['NumRooms'] = pd.to_numeric(data['NumRooms'], errors='coerce')
print(data)
```

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

[2]:
```
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
```

get_dummies를 통해 열의 값에 따른 새로운 열을 생성할 수 있다.

[3]:
```
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
```

```
[4]: import torch

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

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

to_numpy를 사용하면 flase는 0으로, true는 1로 변환된다.

2.3. Linear Algebra

```
[1]: import torch

     x = torch.tensor(3.0)
     y = torch.tensor(2.0)
     x + y, x * y, x / y, x**y
```

```
[1]: (tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

```
[2]: x = torch.arange(3)
     x
```

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

```
[3]: x[2]
```

```
[3]: tensor(2)
```

```
[4]: len(x)
```

```
[4]: 3
```

```
[5]: x.shape
```

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

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

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

```
[7]: A.T
```

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

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

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

'A == A.T' 연산은 A 매트릭스 전체가 AT 매트릭스 전체와 동일한지를 판단하지 않고, 각 행과 열의 일치 여부를 모두 따져서 반환한다. 이를 통해 매트릭스 간 비교 연산은 같은 위치의 값들마다 이뤄짐을 알 수 있다.

```
[9]: torch.arange(24).reshape(2, 3, 4)

[9]: 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]]])
```

```
[10]: torch.arange(23).reshape(2, 3, 4)

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[10], line 1
----> 1 torch.arange(23).reshape(2, 3, 4)

RuntimeError: shape '[2, 3, 4]' is invalid for input of size 23
```

reshape는 길이가 맞지 않으면 오류가 발생한다.

```
[10]: A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
      B = A.clone()
      A, A + B

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

'B = A.clone()'과 같이 clone을 이용하면 새 메모리를 할당하면서 복사할 수 있다.

```
[11]: A * B

[11]: tensor([[ 0.,  1.,  4.],
              [ 9., 16., 25.]])

[12]: a = 2
      X = torch.arange(24).reshape(2, 3, 4)
      a + X, (a * X).shape

[12]: (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]))

[13]: x = torch.arange(3, dtype=torch.float32)
      x, x.sum()

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

[14]: A.shape, A.sum()

[14]: (torch.Size([2, 3]), tensor(15.))

[15]: A.shape, A.sum(axis=0).shape, A.sum(axis=0)

[15]: (torch.Size([2, 3]), torch.Size([3]), tensor([3., 5., 7.]))

[16]: A.shape, A.sum(axis=1).shape, A.sum(axis=1)

[16]: (torch.Size([2, 3]), torch.Size([2]), tensor([ 3., 12.]))
```

cat에서 dim=0이 row 방향, dim=1이 column 방향이었듯이 sum도 axis=0은 같은 row에 있는 값들을 더하고, axis=1은 같은 column에 있는 값들을 더한다.

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

```
[17]: tensor(True)
```

```
[18]: A.mean(), A.sum() / A.numel()
```

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

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

```
[19]: (tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

```
[20]: sum_A = A.sum(axis=1, keepdims=True)
      sum_A, sum_A.shape
```

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

```
[21]: A / sum_A
```

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

```
[22]: A.cumsum(axis=0)
```

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

```
[23]: y = torch.ones(3, dtype = torch.float32)
      x, y, torch.dot(x, y)
```

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

```
[24]: torch.sum(x * y)
```

```
[24]: tensor(3.)
```

```
[25]: A.shape, x.shape, torch.mv(A, x), A@x
```

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

```
[26]: B = torch.ones(3, 4)
      torch.mm(A, B), A@B
```

```
[26]: (tensor([[ 3.,  3.,  3.,  3.],
               [12., 12., 12., 12.]]),
       tensor([[ 3.,  3.,  3.,  3.],
               [12., 12., 12., 12.]]))
```

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

```
[27]: tensor(5.)
```

```
[28]: torch.abs(u).sum()
```

```
[28]: tensor(7.)
```

```
[29]: torch.norm(torch.ones((4, 9)))
```

```
[29]: tensor(6.)
```

Hadamard product, dot product, 매트릭스-벡터/매트릭스 곱셈 등을 모두 함수 또는 기호로 계산할 수 있다.

2.5. Automatic Differentiation

```
[1]: import torch

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

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

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

x.requires_trad(True)를 통해 x의 기울기를 새 메모리를 할당하지 않고도 추적할 수 있다.

```
[3]: y = 2 * torch.dot(x, x)
     y
```

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

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

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

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

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

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

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

'y.backward()'는 y를 x에 대해 미분하여 기울기를 계산하고, 그 결과를 x.grad에 저장한다. 또한 'x.grad.zero_()'를 이용하여 기울기를 초과할 수 있다.

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

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

'y.backward(gradient=torch.ones(len(y)))'는 y의 각 요소에 1을 곱한 상태로 backward 계산을 하며, 'y.sum().backward()'와 동일하다. 'y.sum().backward()'가 더 빠르고 직관적이다.

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

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

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

detach()를 통해 그래프에서 분리한 u는 상수 취급되기 때문에 z를 x에 대해 미분하면 u가 남게 된다.

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

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

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

```
[10]: 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.rand(size=(), requires_grad=True)
      d = f(a)
      d.backward()

      a.grad == d / a
```

```
[10]: tensor(True)
```

함수의 기울기도 계산할 수 있다.

3.1. Linear Regression

```
[1]: %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)

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

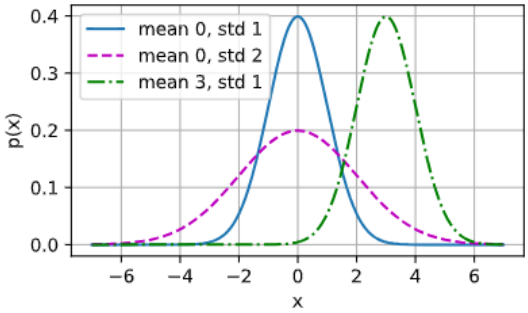
```
[1]: '0.26003 sec'
```

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

```
[2]: '0.00000 sec'
```

'time.time()'으로 현재 시간을 구하고, 이를 통해 연산에 소요된 시간을 계산할 수 있다.

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

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

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



'd2l.plot()'으로 선형 회귀식의 그래프를 그릴 수 있으며, x, y 축의 라벨 또한 지정할 수 있다.

stochastic gradient descent optimizer를 통해 최적화할 수 있다.

선형 회귀는 신경과학에서 많이 사용된다.

3.2. Object-Oriented Design for Implementation

[1]:
```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

[2]:
```
def add_to_class(Class):
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
```

[3]:
```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

[4]:
```
@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

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

'@add_to_class' 데코레이터를 통해 do 함수를 A 클래스에 추가하는 코드이다.

[5]:
```
class HyperParameters:
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

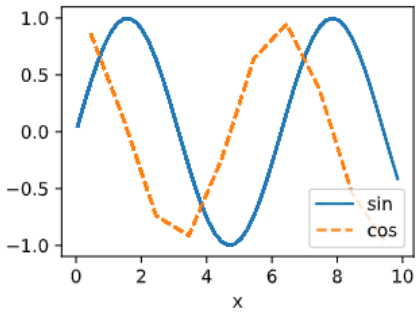
[6]:
```
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

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

c 인자는 저장하지 않는다.

[7]:
```
class ProgressBoard(d2l.HyperParameters):
    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), disply=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

[8]:
```
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)
```



그래프가 애니메이션으로 그려지며, sin 함수는 2 간격으로, cos 함수는 10 간격으로 그려진다.

[9]:
```python
class Module(nn.Module, d2l.HyperParameters):
    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):
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlable = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        raise NotImplementedError
```

모델 클래스를 만들고,

[10]:
```python
class DataModule(d2l.HyperParameters):
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

Training dataset을 불러올 클래스를 만든 후,

[11]:
```
class Trainer(d2l.HyperParameters):
    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
```

Trainer 클래스로 모델 클래스를 학습시킬 수 있다.

3.4. Linear Regression Implementation from Scratch

[12]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

[13]:
```
class LinearRegressionScratch(d2l.Module):
    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)
```

[14]:
```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

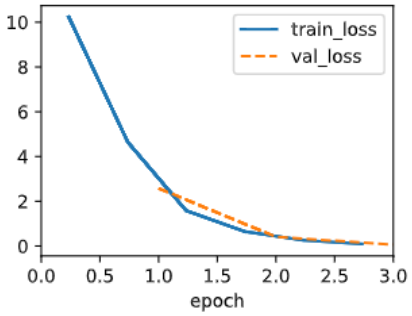
[15]:
```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

[16]:
```
class SGD(d2l.HyperParameters):
    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_()
```

[17]:
```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

[18]:
```
@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
```

[19]:
```
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)
```



[20]:
```
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([ 0.1169, -0.2309])
error in estimating b: tensor([0.2448])
```

선형 회귀 모델을 가중치 [2, -3.4], 편향 4.2를 가진 트레이닝 데이터셋으로 epoch=3만큼 반복하여 학습시키고, 훈련 손실(train loss)과 검증 손실(val loss)을 기록하여 그래프로 시각화 해준다. 학습이 반복될수록 손실이 감소하는 것을 그래프로 확인할 수 있다.

4.1. Softmax Regression

분류는 입력 데이터가 어느 범주(클래스)에 속하는지 예측하는 것이다. 분류의 방법으로는 선형 모델, 소프트맥스, 벡터화가 있다. 이 중 소프트맥스는 nonnegativity와 각 클래스의 확률값의 합

이 1이 되는 것을 보장한다.

분류의 손실 함수에는 Log-Likelihood와 Softmax and Cross-Entropy Loss가 있다. 소프트맥스는 확률을 계산하고, 크로스 엔트로피가 확률이 실제와 얼마나 차이 나는지 측정한다.

4.2. The Image Classification Dataset

[1]:
```
%matplotlib inline
import time
import torch
import torchvision
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

[2]:
```
class FashionMNIST(d2l.DataModule):
    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)
```

[3]:
```
data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

[3]:
```
(60000, 10000)
```

[4]:
```
data.train[0][0].shape
```

[4]:
```
torch.Size([1, 32, 32])
```

[5]:
```
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

[6]:
```
@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)
```

[7]:
```
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

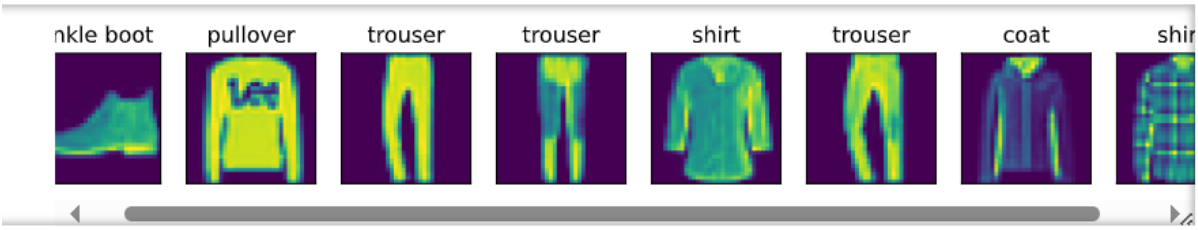
64개의 이미지로 구성된 미니배치를 불러온다.

[8]:
```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

[8]:
```
'24.98 sec'
```

```
[9]: def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
         raise NotImplementedError
```

```
[10]: @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)
```



visualize 메서드를 만들어 분류 이미지를 시각화 할 수 있다.

4.3. The Base Classification Model

```
[1]: import torch
     from d2l import torch as d2l
```

```
[2]: class Classifier(d2l.Module):
         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)
```

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

```
[4]: @d2l.add_to_class(Classifier)
     def accuracy(self, Y_hat, Y, averaged=True):
         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
```

손실 값과 정확도를 측정하고, 선형 회귀처럼 stochastic gradient descent optimizer를 이용해 최적화할 수 있다.

4.4. Softmax Regression Implementation from Scratch

```
[1]: import torch
     from d2l import torch as d2l
```

[2]:
```
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)
```

[2]:
```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

[3]:
```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition
```

[4]:
```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

[4]:
```
(tensor([[0.1737, 0.2060, 0.1502, 0.2488, 0.2213],
         [0.2109, 0.1567, 0.1149, 0.2827, 0.2348]]),
 tensor([1., 1.]))
```

Softmax로 분류한 확률의 합이 각각 1임을 확인할 수 있다.

[5]:
```
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]
```

[6]:
```
@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)
```

[7]:
```
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]
```

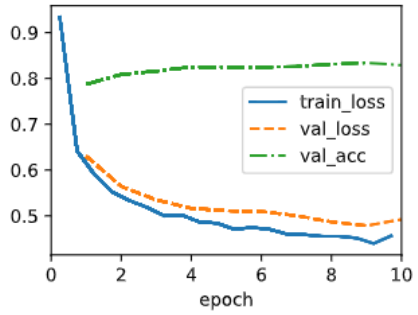
[7]:
```
tensor([0.1000, 0.5000])
```

[8]:
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

[8]:
```
tensor(1.4979)
```

[9]:
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

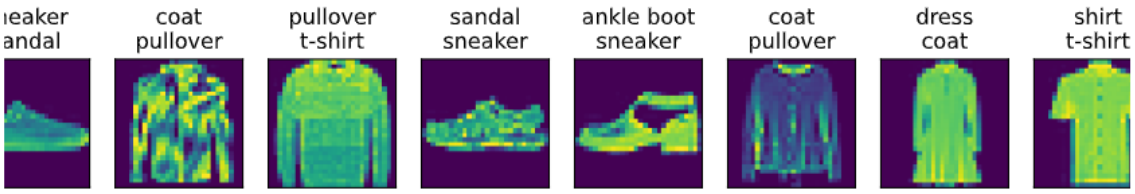
[10]:
```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



[11]:
```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

[11]:
```
torch.Size([256])
```

[12]:
```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a, b in zip(
    data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```



선형회귀 분류와 마찬가지로 시각화 할 수 있다.

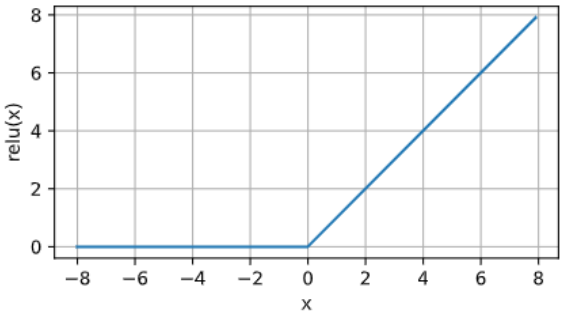
5.1. Multilayer Perceptrons

선형 모델은 입력의 증가가 항상 출력의 증가/감소로 이어진다는 한계를 가진다. 이 한계를 극복하기 위해 숨겨진 레이어를 사용하는 multilayer perceptrons을 활용할 수 있다.

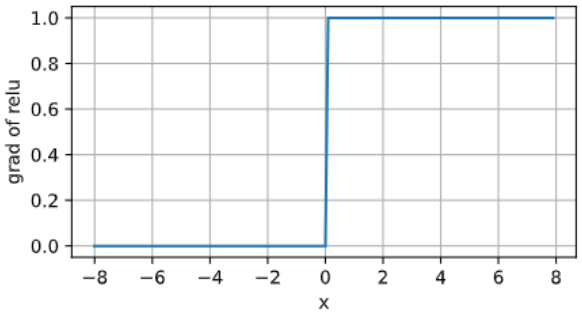
Activate function은 뉴런이 활성화되어야 할지 아닌지를 판단하는데, 그 종류로는 ReLU, Sigmoid, Tanh function이 있다.

[1]:
```
%matplotlib inline
import torch
from d2l import torch as d2l
```

[2]:
```
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))
```



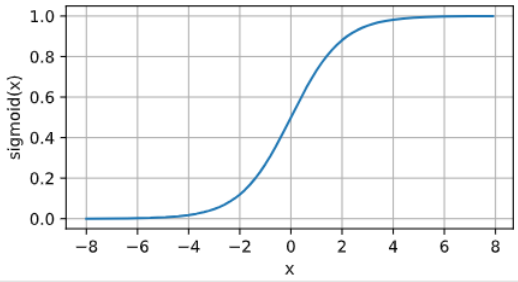
[3]:
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



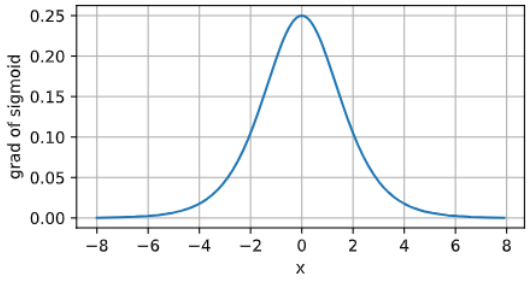
ReLU 함수는 $ReLU(x) = \max(x, 0)$ 식을 가지며, 식에서 알 수 있듯 음수값을 0으로 대체한다. 따라서 ReLU와 그 기울기에 대한 그래프는 위와 같이 그려진다.

또한, ReLU 함수는 parametrized ReLU($pReLU(x) = \max(0, x) + \alpha\min(0, x)$) 등의 변형이 많다.

[4]:
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

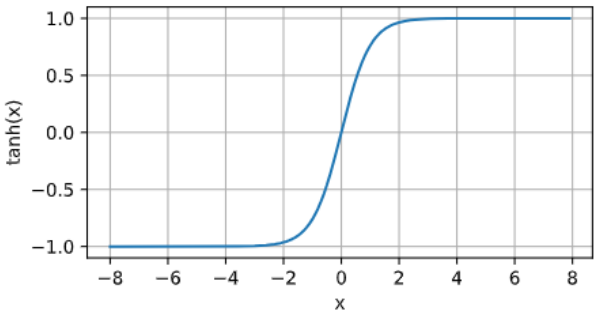


[5]:
```
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

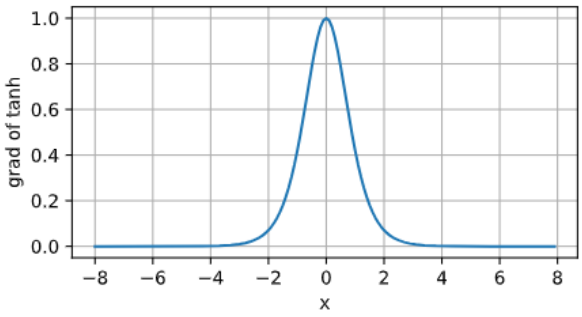


Sigmoid 함수는 $sigmoid(x) = \frac{1}{1+\exp(-x)}$ 식을 가지며, 입력값을 0과 1 사이의 출력값으로 바꾼다. 미분에 따른 기울기는 $sigmoid(x)(1 - sigmoid(x))$이며, 각각의 그래프는 위와 같다.

[6]:
```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



[7]:
```
x.grad.data.zero_()
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



반면 Tanh 함수는 $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$ 식을 가지며, -1과 1 사이의 결과를 반환한다. 기울기는 $1 - tanh^2(x)$이며, 각각의 그래프는 위와 같다.

5.2. Implementation of Multilayer Perceptron

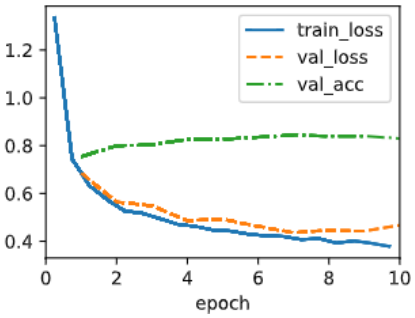
[1]:
```
import torch
from torch import nn
from d2l import torch as d2l
```

[2]:
```
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))
```

[3]:
```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

[4]:
```
@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
```

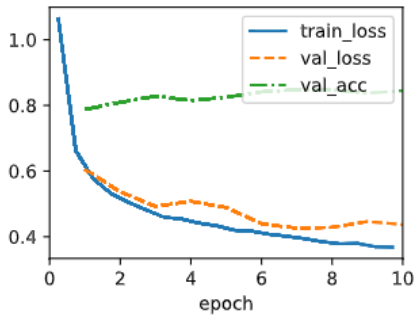
[5]:
```
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



ReLU 함수를 만들어 훈련하고, 그래프로 시각화 할 수 있다.

[6]:
```
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))
```

[7]:
```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



또한 high-level API를 이용해 더 간단하게 작업을 수행할 수도 있다.

5.3. Forward Propagation, Backward Propagation, and Computational Graphs

Forward propagation은 신경망에 대한 중간 변수를 입력 레이어부터 숨겨진 레이어, 출력 레이어까지 순서대로 계산하고 저장하는 것이다. 손실값도 순서대로 계산한다.

Backpropagation은 신경망 매개변수의 기울기를 계산하는 방법이다. 이 방법은 출력 레이어부터 입력 레이어까지 역순으로 미분과 prod 연산을 이용해 계산한다.

두 방법은 신경망을 훈련할 때 모두 사용되며 상호 의존적이다. 그 과정에서 중간 값을 저장해야 하기 때문에 예측(prediction)보다 훈련(training)에 많은 메모리가 필요하다. 이 중간 값은 레이어 수와 배치 크기에 비례한다.