KUBIG 딥러닝 분반 (3주차)

Multilayer Perceptron

딥러닝 분반장 : 김태영, 오화진

01.

Multilayer Perceptron

1. MLP

Recap

이전 장에서 배운 linear transformation에 필요한 가정은?

1. MLP

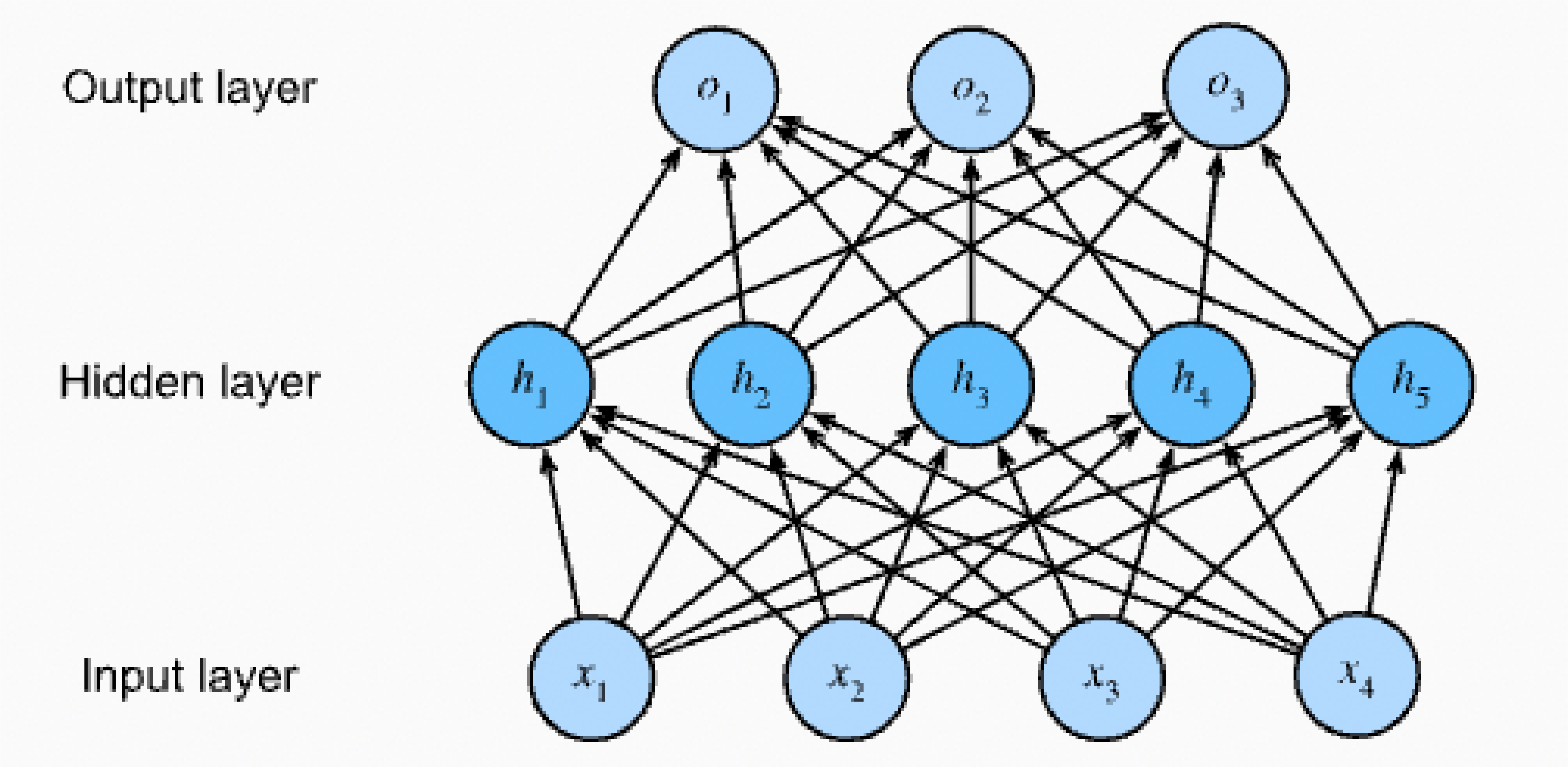
Recap

Linearity = weaker monotonicity

실제 데이터셋들은 linearity가 성립되는 경우가 매우 적음!

1. MLP

MLP = 하나 이상의 hidden layer를 사용하는 방식
input layer = representation of data
output layer = linear predictor



1. MLP

Linear to Nonlinear

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Add one more layer

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}.$$

여전히 affine function... 여전히 linearity를 가정해야 함...
어떤 function이 더 필요하다!

1. MLP

Linear to Nonlinear

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

Activation function = σ

, $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$,

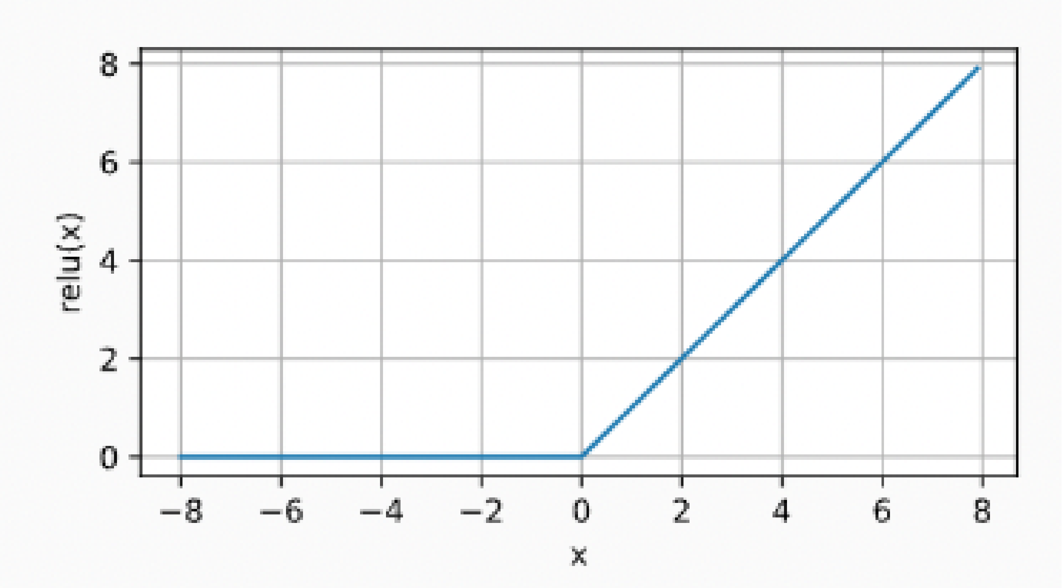
Nonlinear!

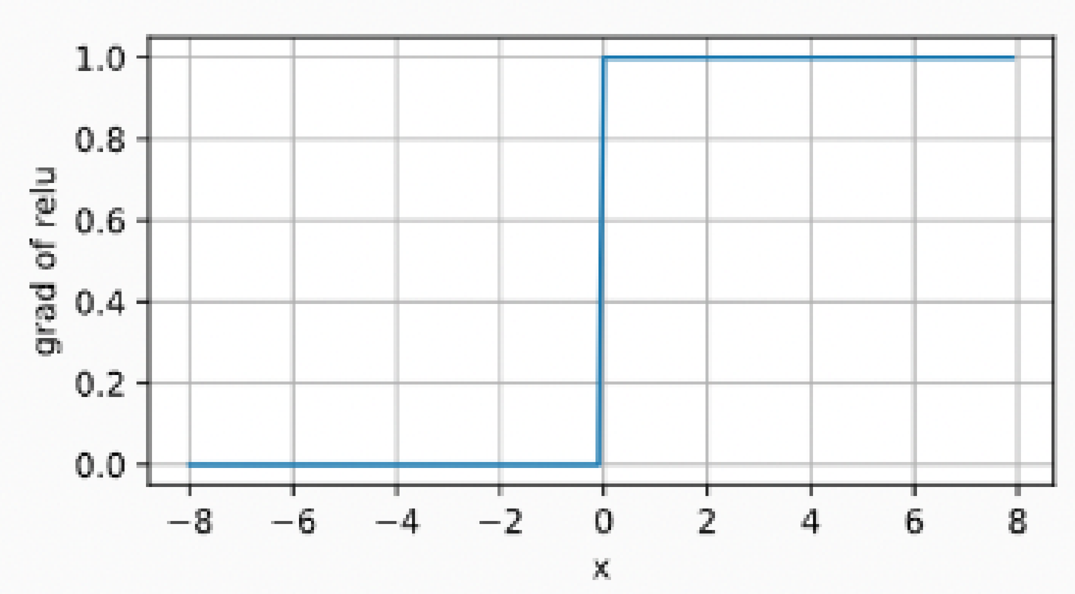
1. MLP

Activation Functions - ReLU
ReLU(x) = max(0, x)

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5,
2.5))
```

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



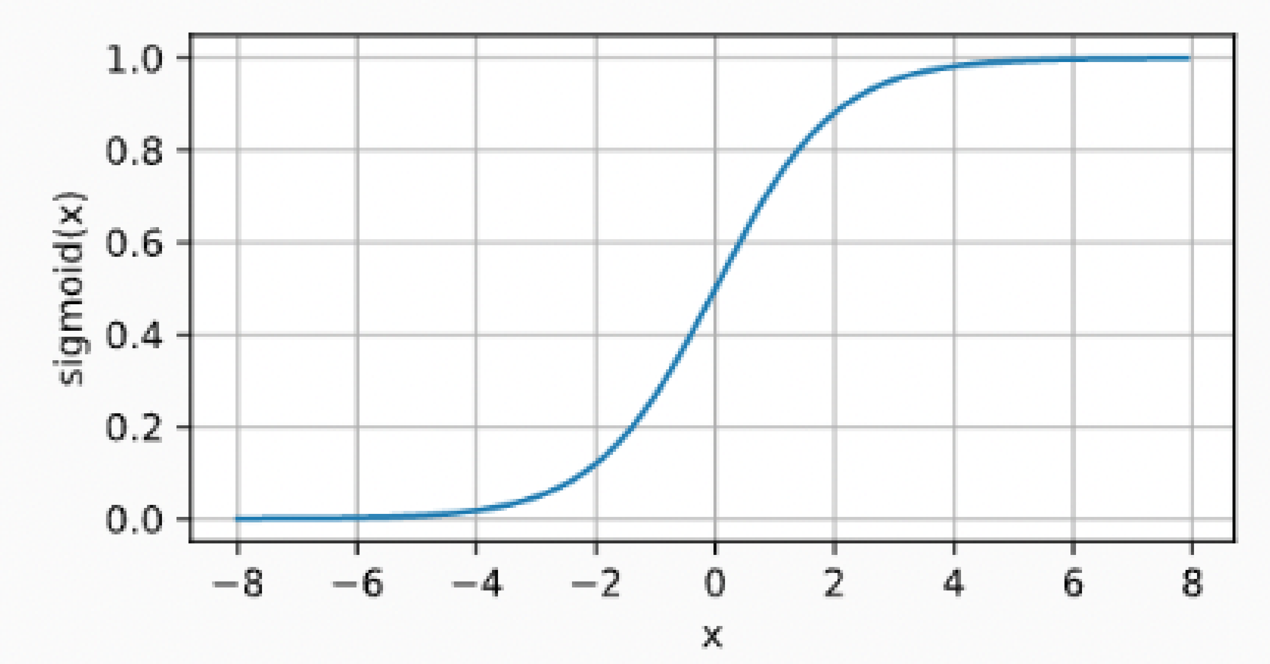


1. MLP

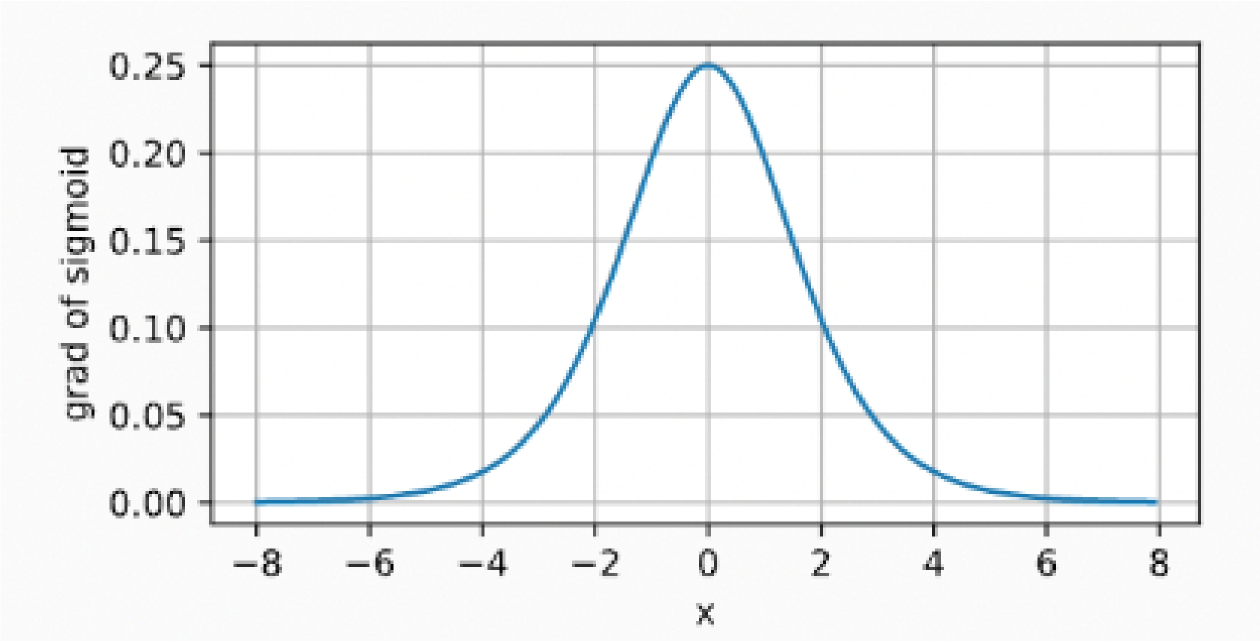
Activation Functions - Sigmoid

Sigmoid(x) = 1 / (1+exp(-x))

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




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

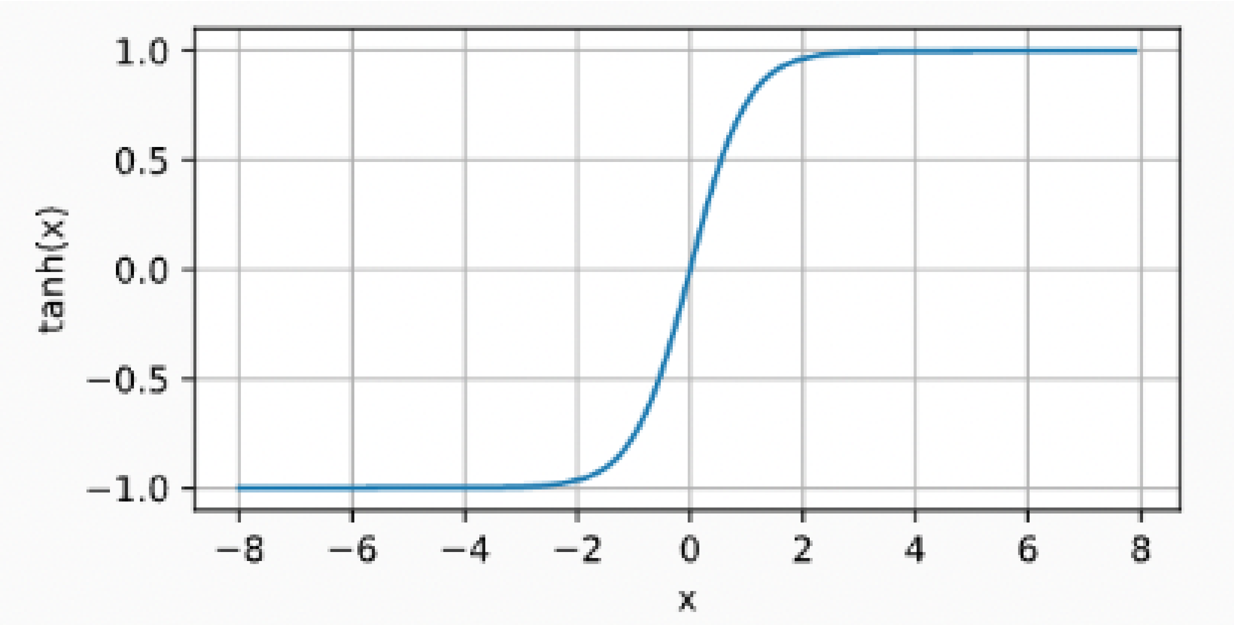


1. MLP

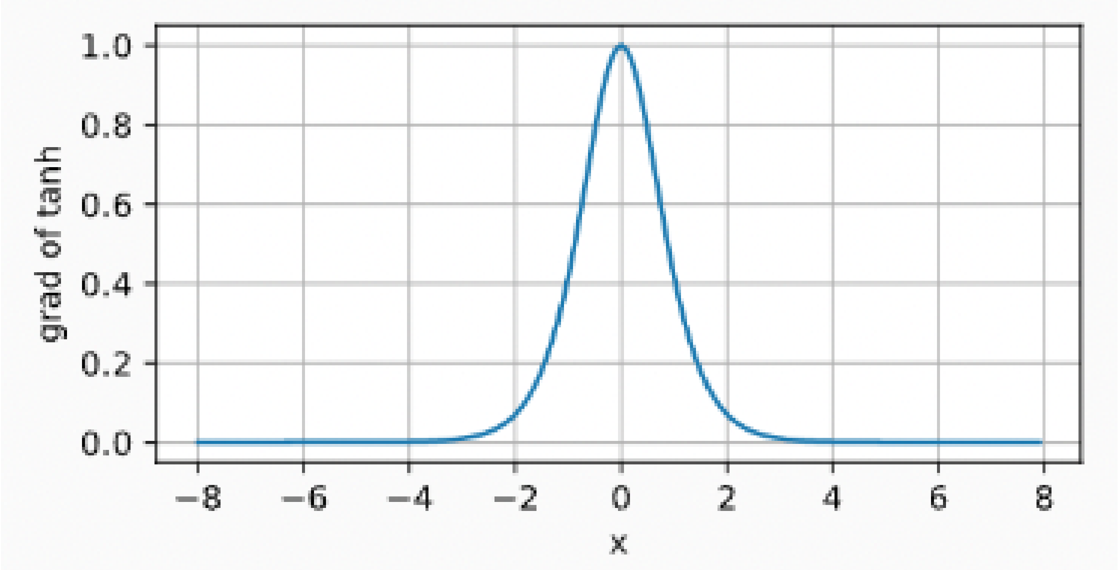
Activation Functions - Tanh

tanh(x) = (1-exp(-2x)) / (1+exp(-2x))

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




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



1. MLP

Scratch

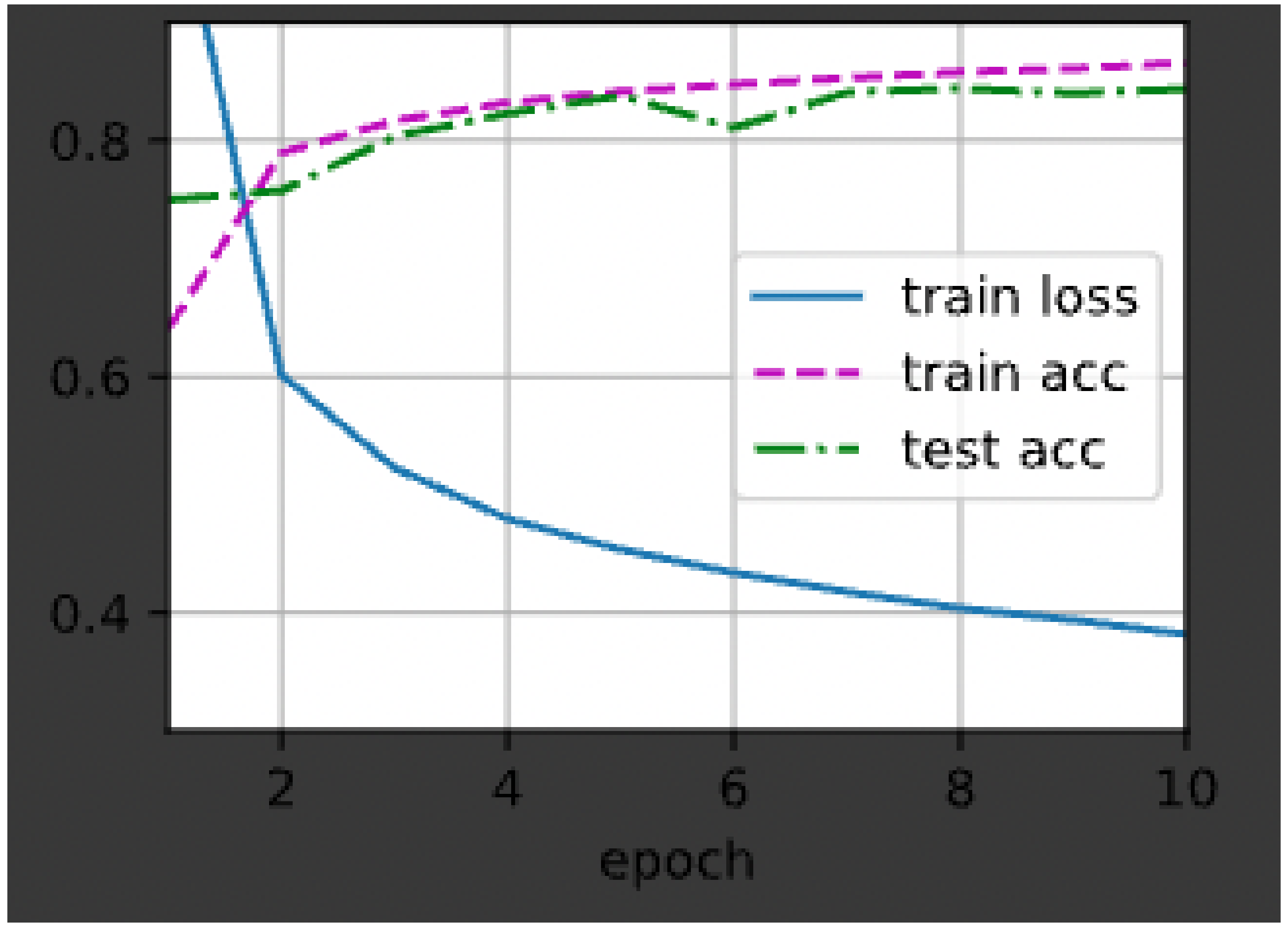
```python
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 256
train_iter, test_iter =
d2l.load_data_fashion_mnist(batch_size)

num_inputs, num_outputs, num_hiddens = 784, 10, 256

# Parameter Initialization, 2 Layers
# torch.randn : tensor filled with random numbers form a
standard normal distribution

W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens,
requires_grad= True)*0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad =
True))
W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs,
requires_grad= True)*0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad =
True))

params = [W1, b1, W2, b2]
```

```python
# Activation Function : ReLU
def relu(X):
  # zeros_like : tensor filled woth the scalar value 0 with
the same size as input
  a = torch.zeros_like(X)
  return torch.max(X,a)

# Model
def net(X):
  X = X.reshape((-1, num_inputs))
  H = relu(X@W1 + b1) # @ : matrix multiplication
  return (H@W2 + b2)

# Loss function
loss = nn.CrossEntropyLoss(reduction = "none")
```

1. MLP

Scratch

```
# Training

num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr = lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
updater)
```



1. MLP

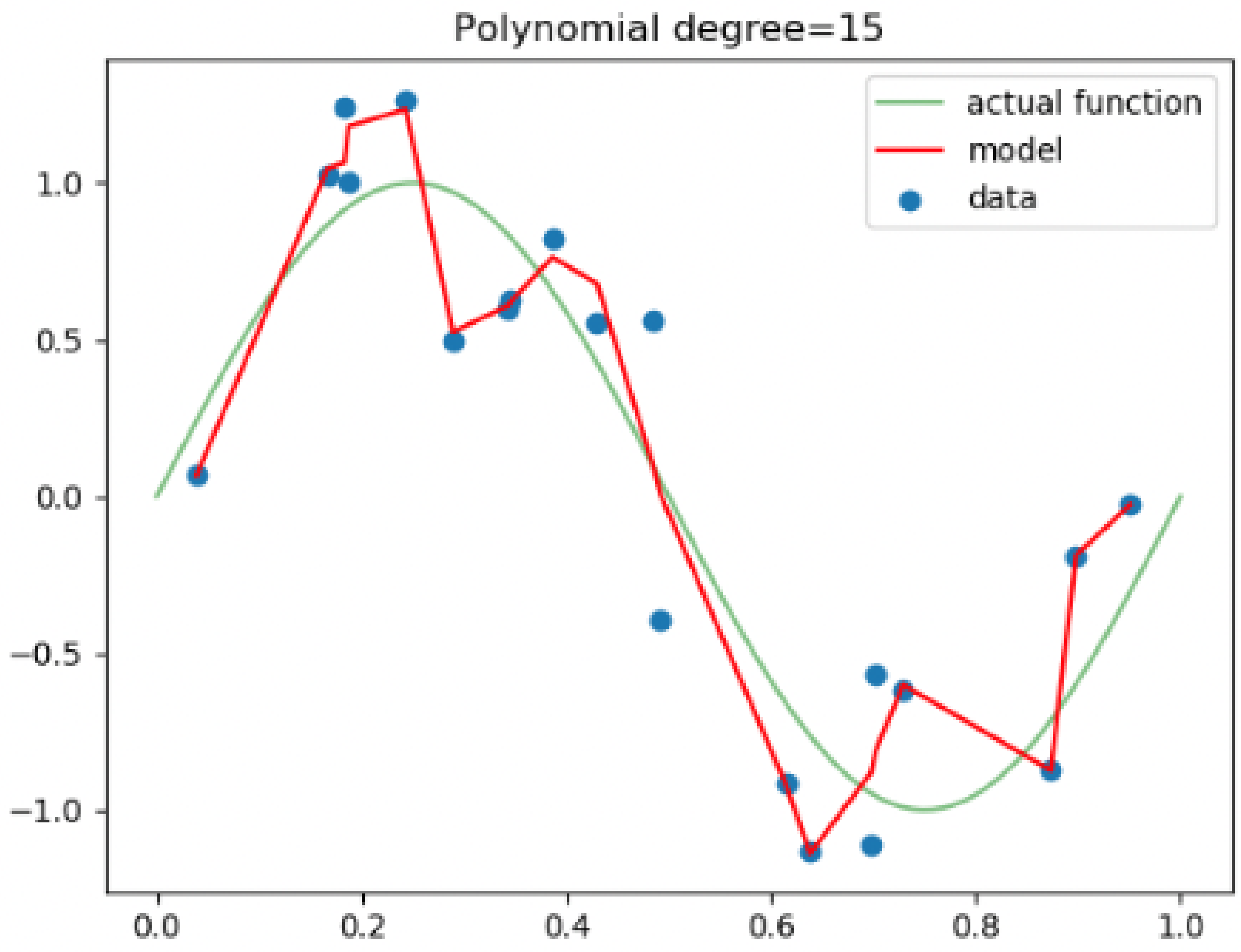
API

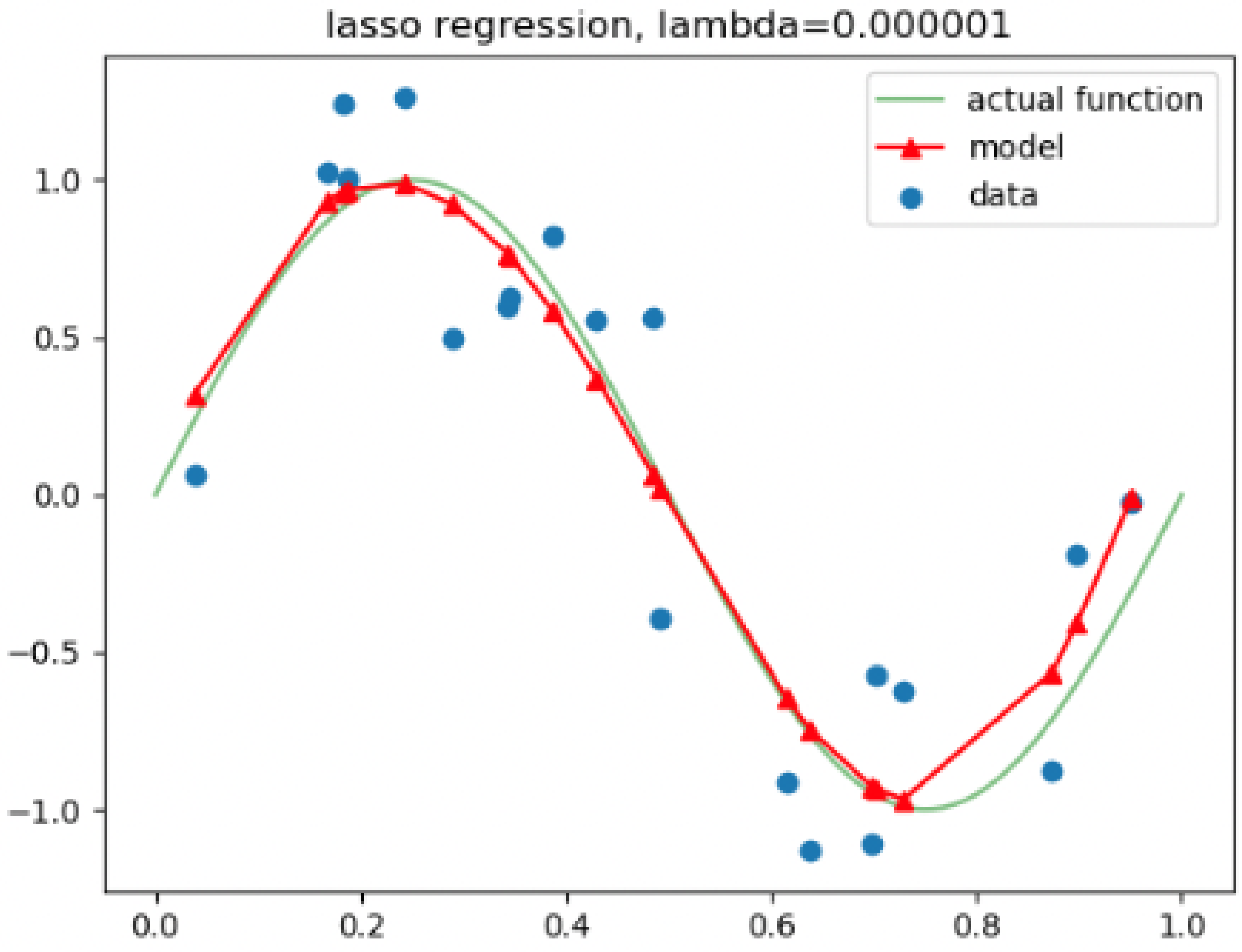
```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Linear(256, 10))

def init_weights(m):
  if type(m) == nn.Linear:
    ## nn.init.normal_ : fills the inut Tensor with values
drawn form the normal distribution
    nn.init.normal_(m.weight, std = 0.01)

net.apply(init_weights)
```

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss(reduction = 'none')
trainer = torch.optim.SGD(net.parameters(), lr = lr)

train_iter, test_iter =
d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
trainer)
```

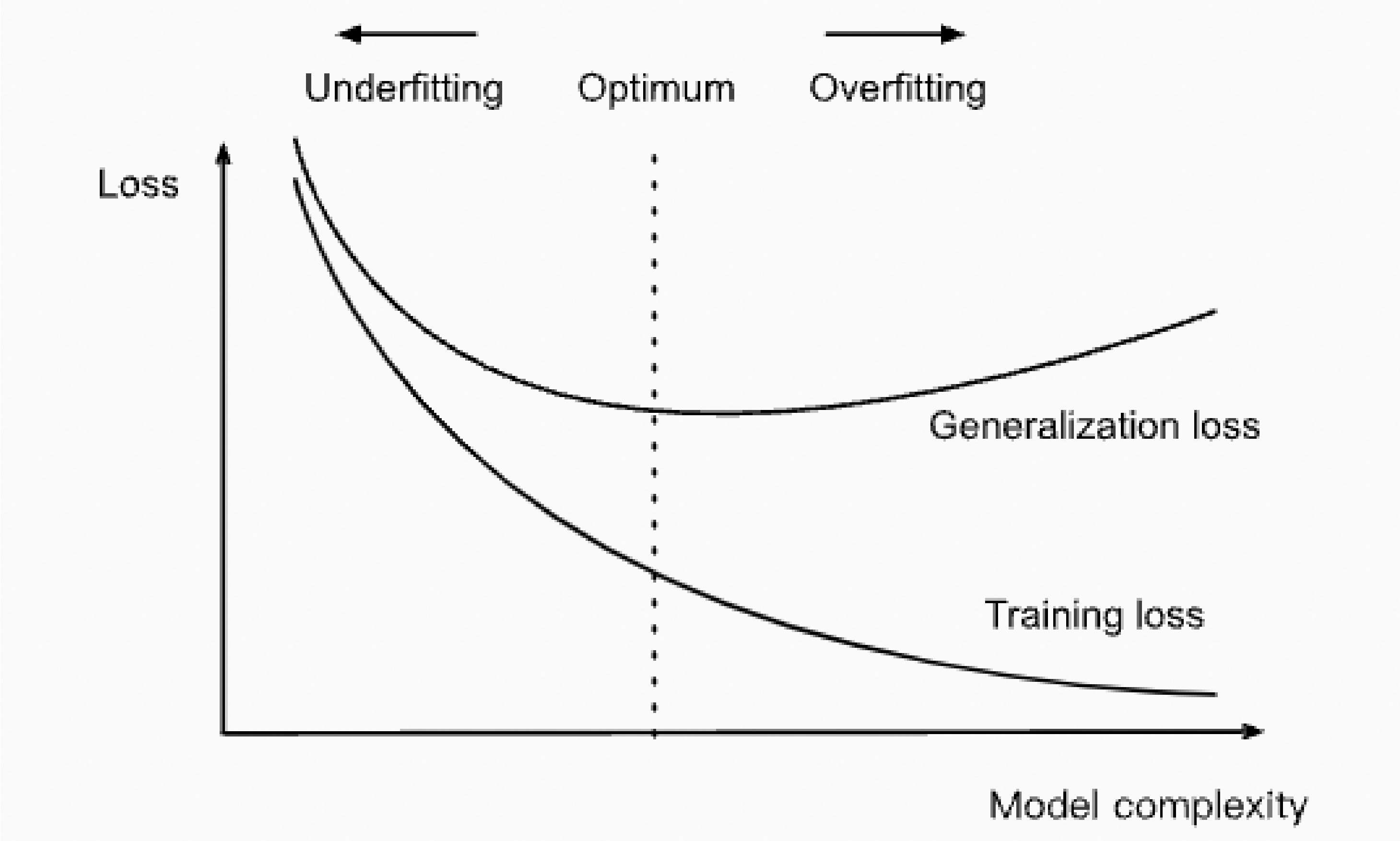
02. Weight Decay

2. Weight Decay





2. Weight Decay



1. generalization error는 training error로 추정될 수 없으며 training loss만 최소화하고자 하는 것은 반드시 generalization error도 함께 최소화하지 않는다. 따라서 머신러닝 알고리즘은 overfitting에 주의해야 한다.

2. validation set을 통해 generalization error를 측정한다.

3. underfitting은 모델이 training error를 줄이지 못함을 말한다. 위 사진처럼 overfitting은 generalization loss가 training loss보다 클 경우를 말한다.

4. overfitting을 막기 위해서는 충분한 양질의 데이터를 사용해야 한다.

2. Weight Decay

Regularization을 통해 Overfitting을 막자

Loss function(train error)의 최소화만을 목표로 한다면?

>> generalization error가 나빠짐
>> 특정 weight의 값만 매우 커짐

penalty를 줘서 특정 weight만이 커지는 것을 막자

< Goal >
"Minimizing the prediction loss on the training labels"

"Minimizing the sum of prediction loss and penalty term"

2. Weight Decay

Regularization을 통해 Overfitting을 막자

Lp norm

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

L1 norm

$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|, \text{ where } (\mathbf{p},\mathbf{q}) \text{ are vectors } \mathbf{p} = (p_1, p_2, \dots, p_n) \text{ and } \mathbf{q} = (q_1, q_2, \dots, q_n)$$

L2 norm

$$\|\boldsymbol{x}\|_2 := \sqrt{x_1^2 + \cdots + x_n^2}.$$

2. Weight Decay

L1 Loss

$$L = \sum_{i=1}^{n} |y_i - f(x_i)|$$

어떤 regularization 방식이 outlier에 더 민감한가?
= Which one is less robust to outlier?

L2 Loss

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

2. Weight Decay

L1 Loss

$$L = \sum_{i=1}^{n} |y_i - f(x_i)|$$

L2 Loss

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

$$a = (0.3, -0.3, 0.4)$$
$$b = (0.5, -0.5, 0)$$

$$||a||_1 =$$
$$||b||_1 =$$

$$||a||_2 =$$
$$||b||_2 =$$

?

2. Weight Decay

L1 Loss

$$L = \sum_{i=1}^{n} |y_i - f(x_i)|$$

L2 Loss

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

$$a = (0.3, -0.3, 0.4)$$
$$b = (0.5, -0.5, 0)$$

$$||a||_1 = |0.3| + |-0.3| + |0.4| = 1$$
$$||b||_1 = |0.5| + |-0.5| + |0| = 1$$

$$||a||_2 = \sqrt{0.3^2 + (-0.3^2) + 0.4^2} = 0.583095$$
$$||b||_2 = \sqrt{0.5^2 + (-0.5^2) + 0^2} = 0.707107$$

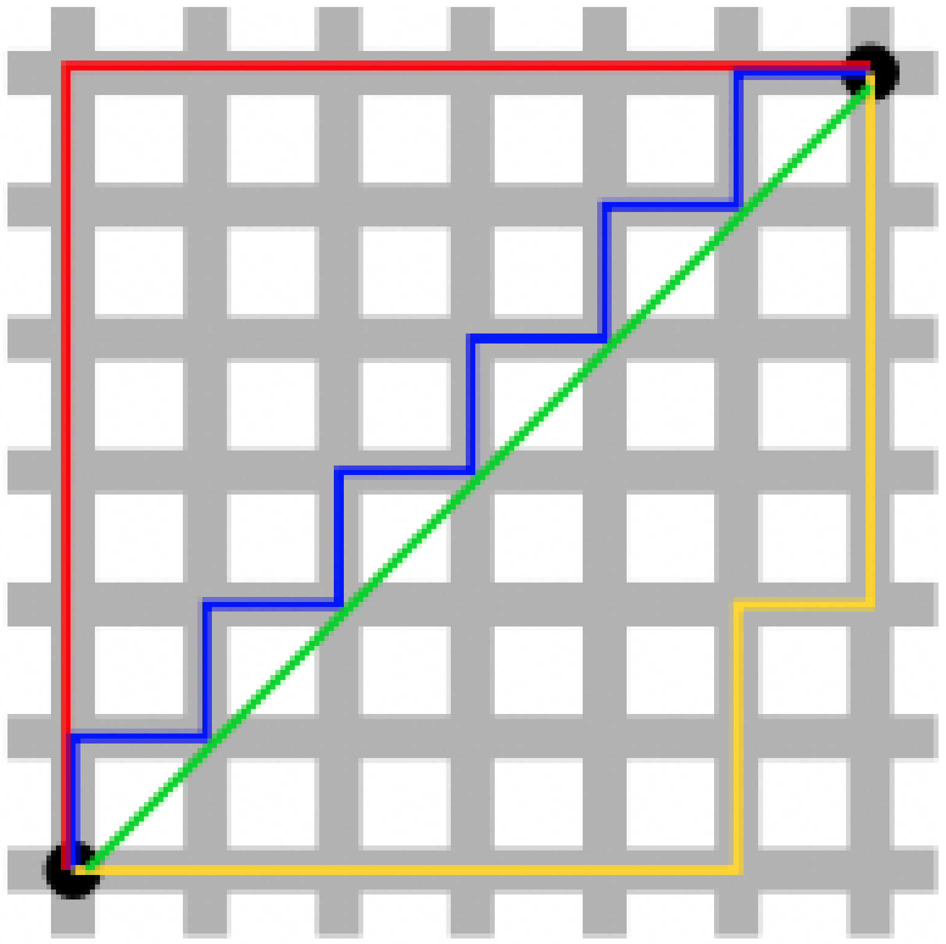
2. Weight Decay

L1 Loss

$$L = \sum_{i=1}^{n} |y_i - f(x_i)|$$

L2 Loss

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$



Green = L2
[Red, Blue, Yellow] = L1

[Red, Blue, Yellow] 중에 하나를 선택
= Feature selection

2. Weight Decay

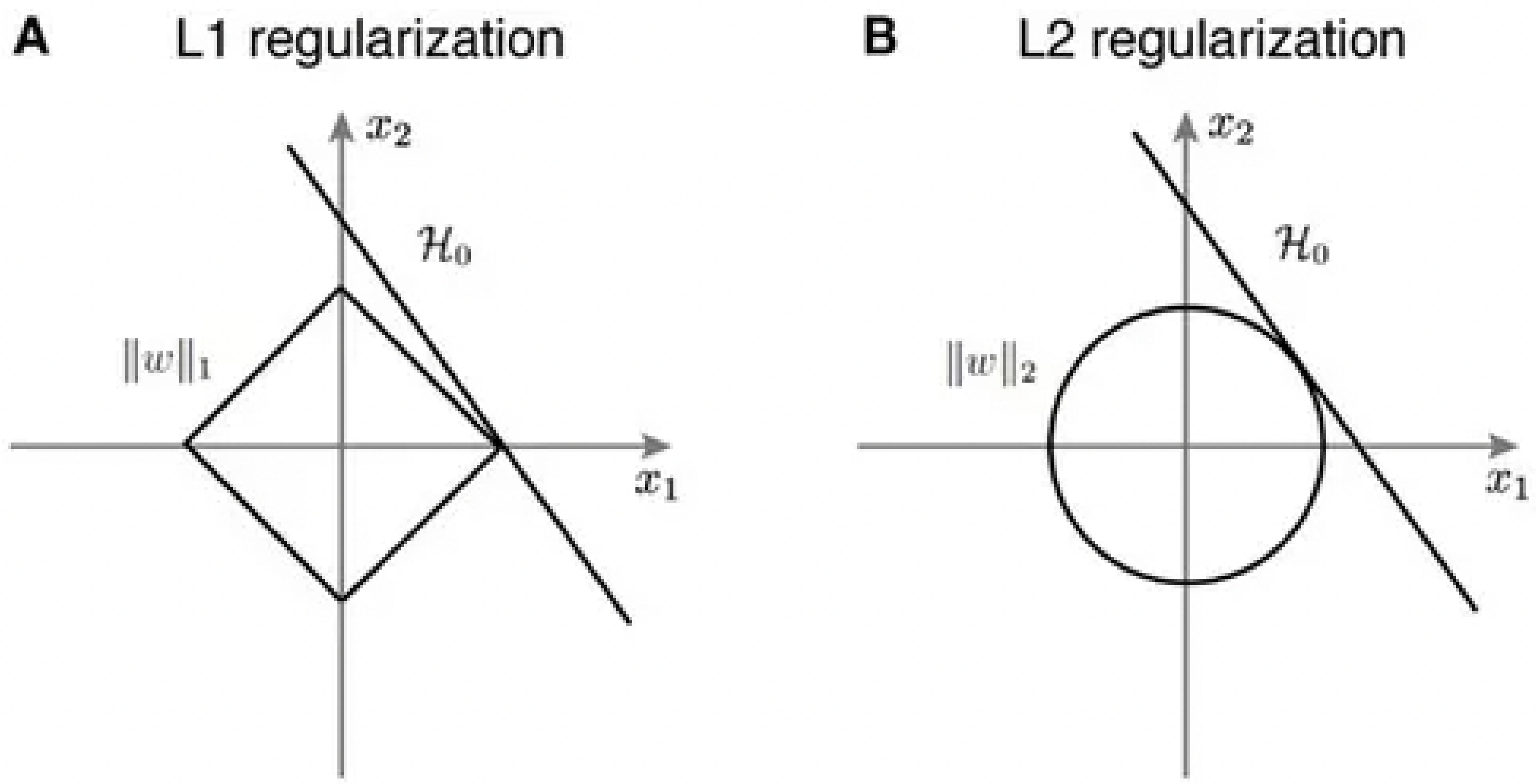
L1 Regularization (Lasso Regression)

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y}_i) + \frac{\lambda}{2}|w|\}$$

$L(y_i, \hat{y}_i)$: 기존의 Cost function

L2 Regularization (Ridge Regression)

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y}_i) + \frac{\lambda}{2}|w|^2\}$$



1. Penalty 효과가 크다.
2. derivative 연산이 쉽다.

2. Weight Decay

Scratch

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2).$$

```
# small training dataset, 200 dimensionality

n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = torch.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size,
is_train=False)
```

2. Weight Decay

Scratch

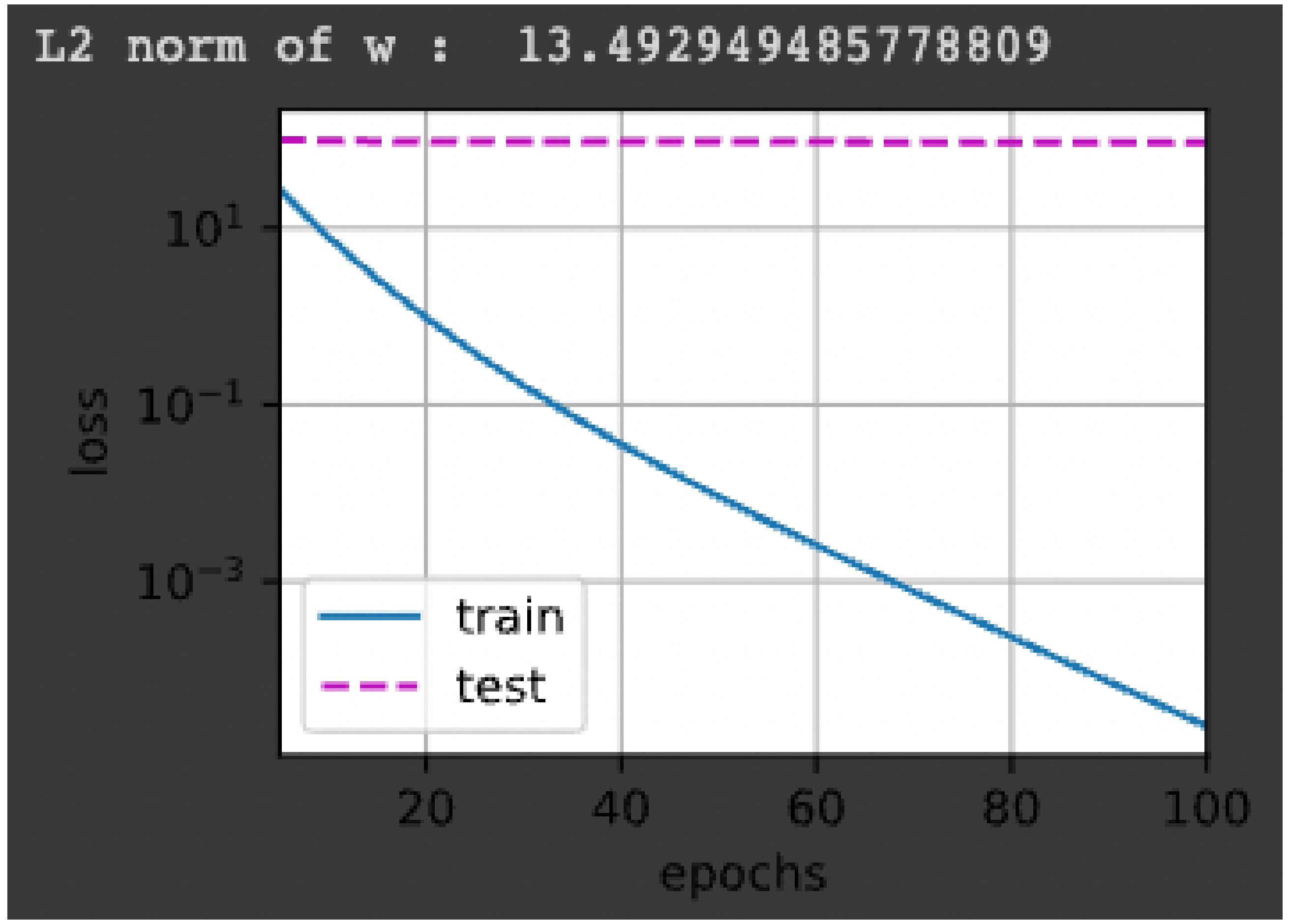
```python
# Initializing

def init_params():
  w = torch.normal(0,1, size = (num_inputs, 1),
requires_grad = True)
  b = torch.zeros(1, requires_grad = True)
  return [w, b]


# L2 norm penalty

def l2_penalty(w):
  return torch.sum(w.pow(2))/2
```

```python
def train(lambd):
  w, b = init_params()
  net, loss = lambda X : d2l.linreg(X, w, b),
d2l.squared_loss
  num_epochs, lr = 100, 0.003
  animator = d2l.Animator(xlabel='epochs', ylabel='loss',
yscale='log',
                            xlim=[5, num_epochs], legend=
['train', 'test'])

  for epoch in range(num_epochs):
    for X, y in train_iter:
      # broadcast the the L2 norm penalty
      l = loss(net(X), y) + lambd * l2_penalty(w)
      l.sum().backward()
      d2l.sgd([w,b], lr, batch_size)
    if (epoch + 1)%5 == 0:
      animator.add(epoch+1, (d2l.evaluate_loss(net,
train_iter, loss),
                             d2l.evaluate_loss(net,
test_iter, loss)))
  print("L2 norm of w : ", torch.norm(w).item())
```

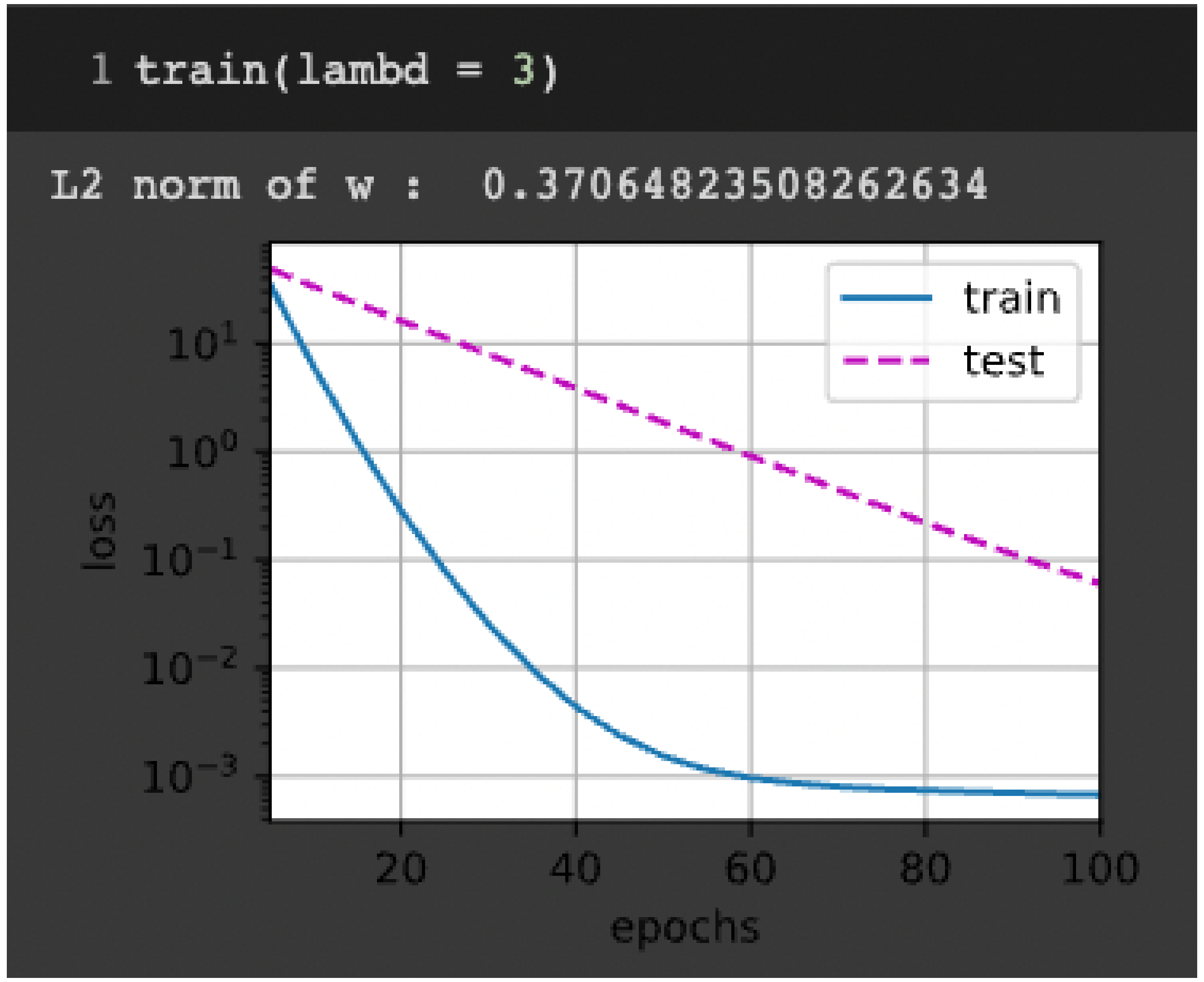
2. Weight Decay

Scratch

lambda = 0



lambda = 3



2. Weight Decay

```
# specify weight decay hyperparameter directly through
weight_decay
# not decay the bias (Pytorch decays both originally)

def train_concise(wd):
  net = nn.Sequential(nn.Linear(num_inputs, 1))
  for param in net.parameters():
    param.data.normal_()
  loss = nn.MSELoss(reduction = 'none')
  num_epochs, lr = 100, 0.003
  trainer = torch.optim.SGD([
        {"params":net[0].weight,'weight_decay': wd},
        {"params":net[0].bias}], lr=lr)
  animator = d2l.Animator(xlabel='epochs', ylabel='loss',
yscale='log',
                          xlim=[5, num_epochs], legend=
['train', 'test'])
  for epoch in range(num_epochs):
      for X, y in train_iter:
          trainer.zero_grad()
          l = loss(net(X), y)
          l.mean().backward()
          trainer.step()
      if (epoch + 1) % 5 == 0:
          animator.add(epoch + 1, (d2l.evaluate_loss(net,
train_iter, loss),
                                   d2l.evaluate_loss(net,
test_iter, loss)))
  print('L2 norm of w:', net[0].weight.norm().item())
```

03.

Dropout

3. Dropout

Linear model

1. feature간 interaction 반영 X
2. positive, negative weight값 만을 규정
3. context 반영 X
4. example > feature : 대부분의 경우 overfitting X
5. High Bias , Low variance

VS

Deep neural network

1. feature간 interaction 반영 O
2. example > feature : overfitting 충분히 발생 가능
3. High Flexibility

3. Dropout

Smoothness = Simplicity

: make function robust to small change in data

↓

Dropout

: inject noise while computing each internal layer during forward propagation

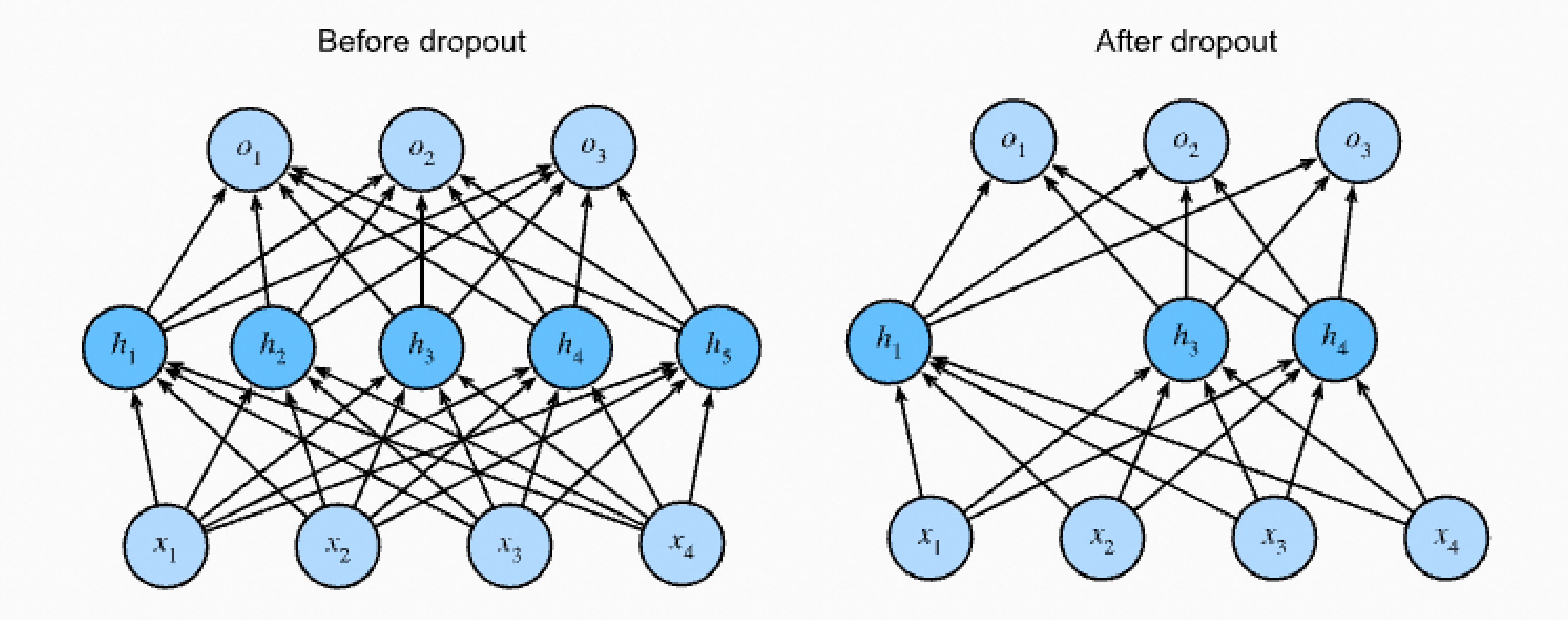
= training 중 몇몇의 neuron을 zeroing(dropout)

= layer간 co-adaptation을 깨는 것

3. Dropout

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

unbiased 방식으로 noise를 더함
(noise를 더한 각 layer의 기댓값) == (noise를 더하지 않은 각 layer의 기댓값)



3. Dropout

```python
def dropout_layer(X, dropout):
  assert 0 <= dropout <= 1
  if dropout == 1 :
    return torch.zeros_like(X) # drop all
  if dropout == 0: # all kept
    return X
  mask = (torch.rand(X.shape) > dropout).float()
  return mask * X / (1.0 - dropout)
```

3. Dropout

```
a = torch.rand(2,3)
a
```

```
tensor([[0.5257, 0.0786, 0.9414],
        [0.8939, 0.3256, 0.3401]])
```

```
a> 0.1
```

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

```
(a> 0.1).float()
```

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

```
X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
print(X)
print(dropout_layer(X, 0.))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1.))
```

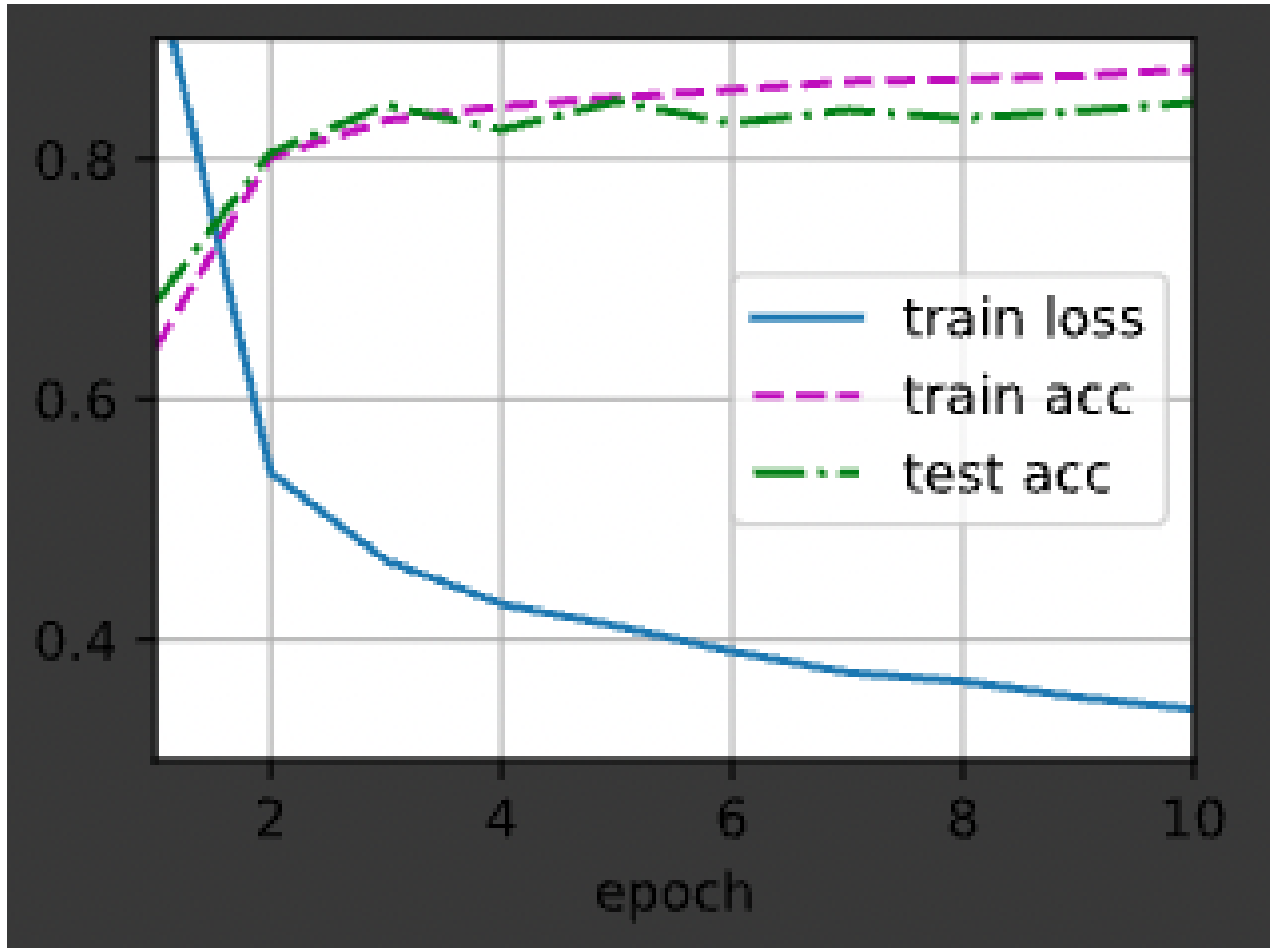
```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  2.,  0.,  0.,  0.,  0., 12., 14.],
        [16.,  0., 20.,  0., 24.,  0., 28., 30.]])
tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

3. Dropout

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784,
10, 256, 256

dropout1, dropout2 = 0.2, 0.5

class Net(nn.Module):
  def __init__(self, num_inputs, num_outputs, num_hiddens1,
num_hiddens2, is_training = True):
    super(Net, self).__init__()
    self.num_inputs = num_inputs
    self.training = is_training
    self.lin1 = nn.Linear(num_inputs, num_hiddens1)
    self.lin2 = nn.Linear(num_hiddens1, num_hiddens2)
    self.lin3 = nn.Linear(num_hiddens2, num_outputs)
    self.relu = nn.ReLU()

  def forward(self, X) :
    H1 = self.relu(self.lin1(X.reshape((-1,
self.num_inputs))))
    # use dropout only when training
    if self.training == True:
      H1 = dropout_layer(H1, dropout1) # add dropout layer
    H2 = self.relu(self.lin2(H1))
    if self.training == True:
      H2 = dropout_layer(H2, dropout2) # add dropout layer
    out = self.lin3(H2)
    return out

net = Net(num_inputs, num_outputs, num_hiddens1,
num_hiddens2)
```

3. Dropout

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = nn.CrossEntropyLoss(reduction = 'none')
train_iter, test_iter =
d2l.load_data_fashion_mnist(batch_size)
trainer = torch.optim.SGD(net.parameters(), lr = lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
trainer)
```



3. Dropout

```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Dropout(dropout1),
                    nn.Linear(256, 256),
                    nn.Dropout(dropout2),
                    nn.Linear(256, 10))

def init_weights(m):
  if type(m) == nn.Linear:
    nn.init.normal_(m.weight, std = 0.01)

net.apply(init_weights)
```

```
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
trainer)
```

04.

Forward & Backward Propagation

4. Propagation

< Forward Propagation >

input : $x \in \mathbb{R}^d$, no bias

$z = W^{(1)} x$ (hidden layer weight parameter = $W^{(1)} \in \mathbb{R}^{h \times d}$)

(intermediate variable $z \in \mathbb{R}^h$)

$\downarrow$

$h = \phi(z)$ (activation function = ϕ)

(hidden variable = h)

$\downarrow$

$o = W^{(2)} h$ (output layer weight = $W^{(2)} \in \mathbb{R}^{q \times h}$)

$\downarrow$ o : length q vector

$L = l(o, y)$ (loss function = l)

(example label = y)

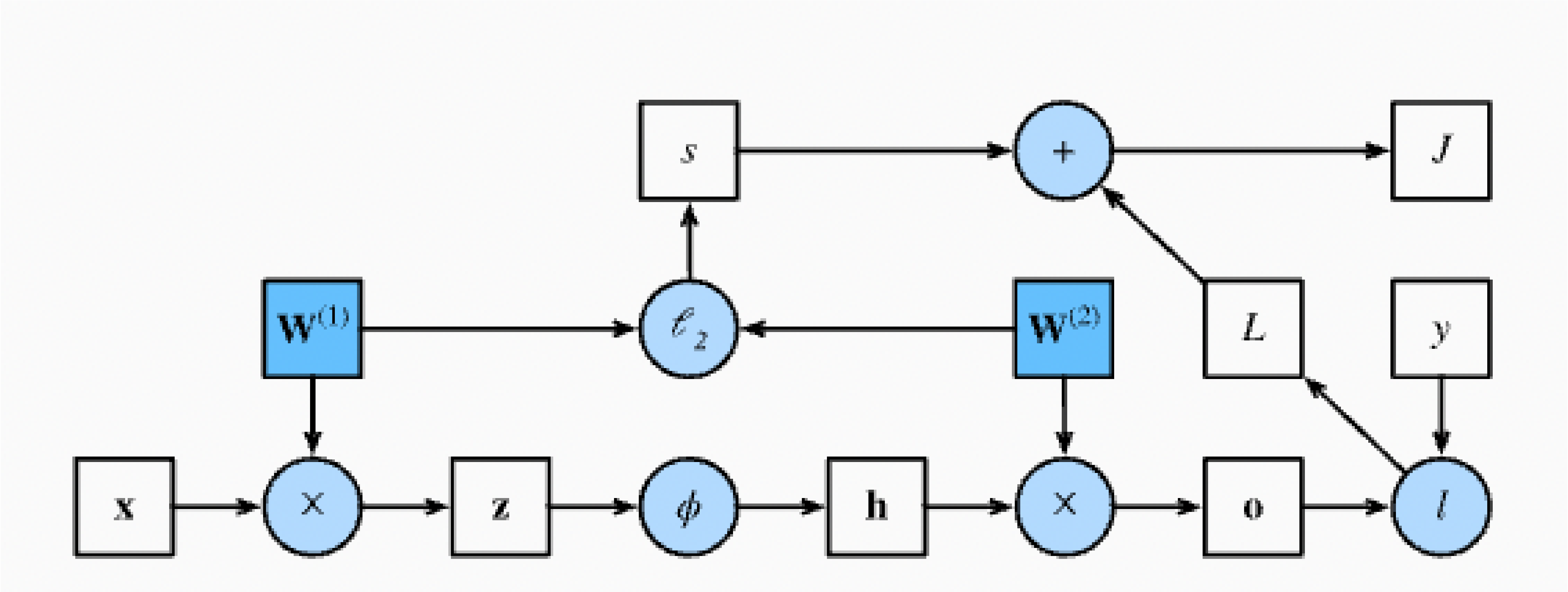
$S = \frac{\lambda}{2} \left(\| W^{(1)} \|_F^2 + \| W^{(2)} \|_F^2 \right)$ $\longrightarrow$ $J = L + S$

(regularization term = S)

($\| \ \|_F$ = Frobenius norm = L_2 norm applied after flattening the matrix)

J = model's regularized loss = objective function

4. Propagation



4. Propagation

<Backward Propagation>

objective : calculate $\partial J/\partial W^{(1)}$, $\partial J/\partial W^{(2)}$

$J = L + S$

$\rightarrow \partial J/\partial L = 1$, $\partial J/\partial S = 1$

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

$$\partial S/\partial W^{(1)} = \lambda W^{(1)} , \frac{\partial S}{\partial W^{(2)}} = \lambda W^{(2)}$$

$$\partial J/\partial W^{(2)} = \text{prod}\left(\partial J/\partial o, \partial o/\partial W^{(2)}\right) + \text{prod}\left(\partial J/\partial S, \partial S/\partial W^{(2)}\right) = \frac{\partial J}{\partial o} h^{T} + \lambda W^{(2)}$$

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

element wise multiplication

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

$$\partial J/\partial W^{(1)} = \text{prod}\left(\partial J/\partial z, \partial z/\partial W^{(1)}\right) + \text{prod}\left(\partial J/\partial S, \frac{\partial S}{\partial W^{(1)}}\right) = \frac{\partial J}{\partial z} X^{T} + \lambda W^{(1)}$$

05.

Numerical Stability & Initialization

5. Stability & Initialization

linear operation을 수행할건데 이 때 어떤 intialization 방법을 수행해야 하는가?
사용하는 activation function에 따라 어떤 intialization을 선택해야 하는가?

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}).$$

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}.$$

5. Stability & Initialization

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}.$$

numerical underflow

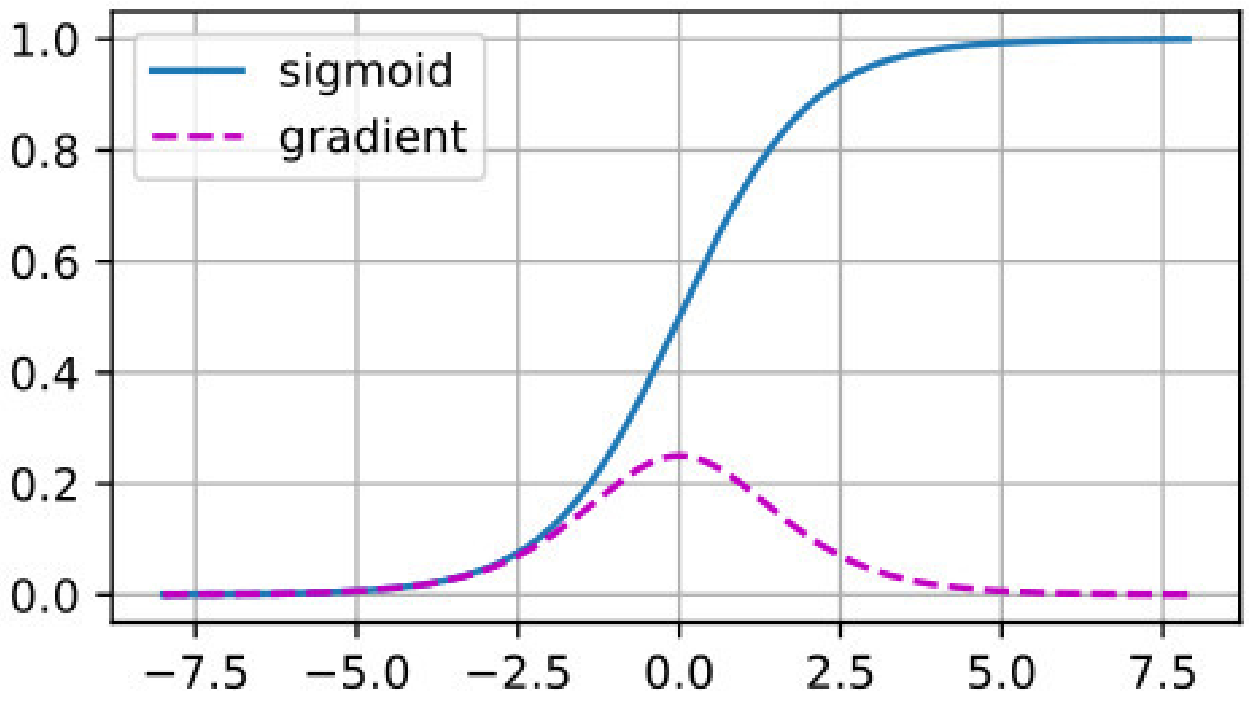
: 너무 많은 확률값들을 곱할 때 생기는 문제

>> representation 어려움
>> gradient 연산 불안정 → Vanishing, Exploding Gradient

5. Stability & Initialization

Vanishing Gradient

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

x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))

d2l.plot(x.detach().numpy(), [y.detach().numpy(),
x.grad.numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



ReLU를 쓰자!

5. Stability & Initialization

Exploding Gradient

```
M = torch.normal(0, 1, size=(4,4))
print('a single matrix \n',M)
for i in range(100):
    M = torch.mm(M,torch.normal(0, 1, size=(4, 4)))

print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[ 0.3888,  0.3659, -0.8204, -0.8018],
        [-0.4795,  2.9719, -0.4333,  0.0644],
        [-0.6021,  0.4666, -0.2969, -0.0676],
        [ 1.3210,  0.1637, -0.2545,  0.4667]])
after multiplying 100 matrices
 tensor([[-1.4178e+24,  3.6789e+25, -3.3382e+25,  3.7610e+24],
        [ 2.3629e+24, -6.1317e+25,  5.5637e+25, -6.2685e+24],
        [-1.9885e+24,  5.1599e+25, -4.6819e+25,  5.2750e+24],
        [ 9.6357e+24, -2.5003e+26,  2.2688e+26, -2.5561e+25]])
```

5. Stability & Initialization

Breaking Symmetry

symmetry input >> output값이 동일 >> update 변경사항 없음

↓

random initialzziation & dropout regularization

5. Stability & Initialization

Xavier Initialization

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}.$$

1

$$\begin{aligned}
E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\
&= 0, \\
\text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\
&= \boxed{n_{\text{in}} \sigma^2} \gamma^2.
\end{aligned}$$

5. Stability & Initialization

Xavier Initialization

Xavier distribution ~ mean = 0, var = $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$.

1. 어떤 output의 분산도 input의 갯수에 영향을 받지 않는다.

2. 어떤 gradient의 분산도 ouput의 갯수에 영향을 받지 않는다.

06.

4주차 코딩과제