Chapter 4 신경망 학습

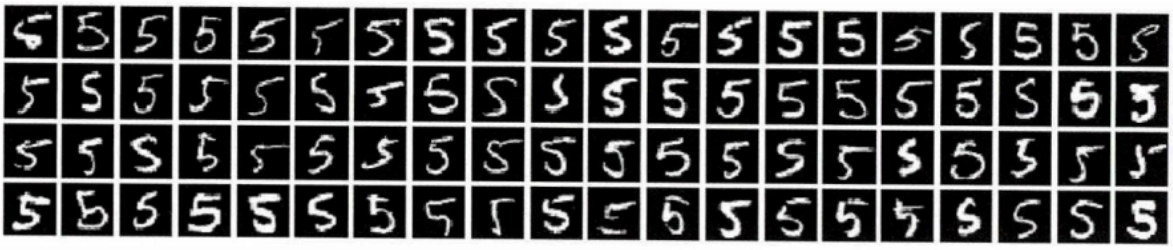
4.1 데이터에서 학습된다!

- 신경망의 특징은 데이터를 보고 학습할수 있다는 점이다.

→따라서 이번 장에서는 신경망 학습(데이터로부터 매개변수의 값을 정하는 방법)에 대해 설명하고 파이썬으로 MNIST 데이터셋의 손글씨 숫자를 학습하는 코드를 구현해본다.

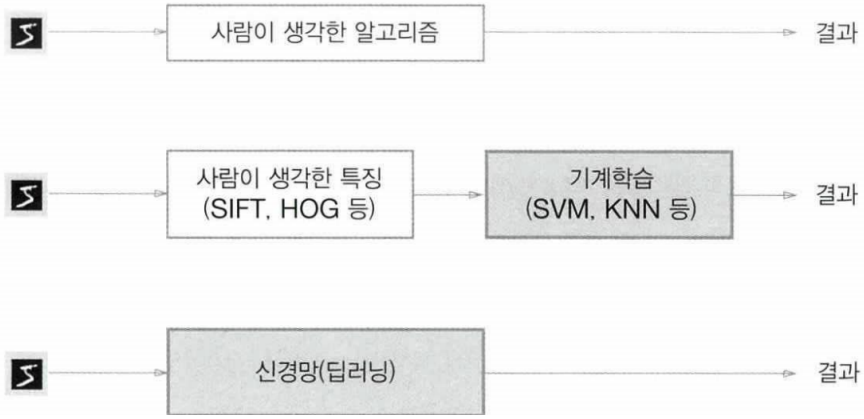
4.1.1 데이터 주도 학습

그림 4-1 손글씨 숫자 '5'의 예 : 사람마다 자신만의 필체가 있다.



- 해당 그림의 이미지만을 보고 5인지 아닌지 판단하는 알고리즘을 구현하는 것은 어려운 일이다.
- 따라서 밑바닥부터 설계하는 대신, 이미지에서 특징(feature)을 추출하고 그 특징의 패턴을 기계학습 기술로 학습하는 방법이 있다.

그림 4-2 규칙을 '사람'이 만드는 방식에서 '기계'가 데이터로부터 배우는 방식으로의 패러다임 전환 : 회색 블록은 사람이 개입하지 않음을 뜻한다.



- 해당 그림과 같이 도식화 시킬수 있고 이러한 딥러닝을 종단간 기계학습(end-to-end machine learning)이라고 한다.

4.1.2 훈련 데이터와 시험 데이터

- 훈련 데이터와 시험 데이터로 나누는 이유 : 범용적으로 사용할 수 있는 모델을 찾기 위해서이다. 즉 한 데이터셋에만 지나치게 최적화된 오버피팅을 피하기 위해서이다.

4.2 손실 함수

- 특정 행동을 평가함에 있어, 신경망은 '하나의 지표 = 손실 함수'를 기준으로 최적의 매개변수를 탐색한다.
- 일반ㅇ적으로 오차제곱합과 교차 엔트로피 오차를 사용한다.

4.2.1 오차제곱함(Sum of squares of error, SEE)

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

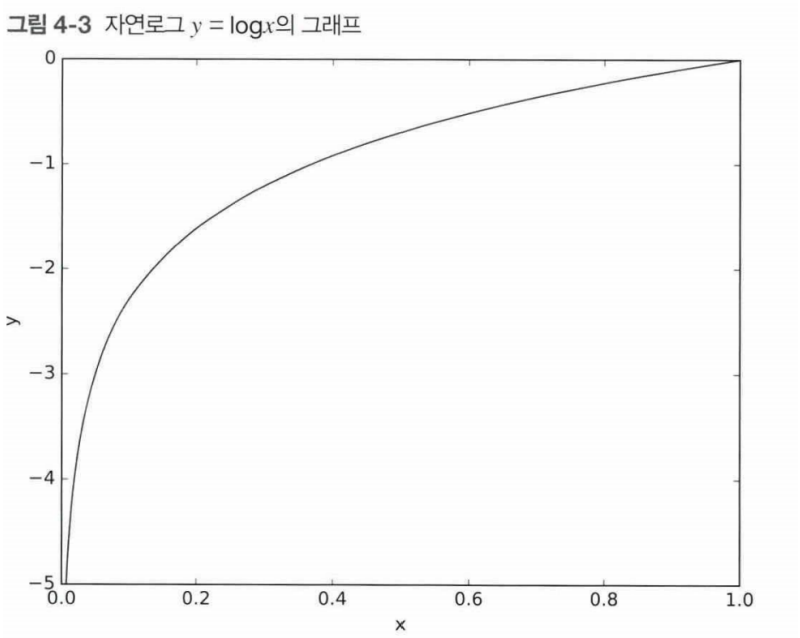
```
# 원-핫 인코딩
import numpy as np
y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
t =[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

def sum_squares_error(y, t):
  return 0.5 * np.sum((y-t) ** 2)

test1 = sum_squares_error(np.array(y), np.array(t))
print(test1)

y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
test2 = sum_squares_error(np.array(y), np.array(t))
print(test2)
```

4.2.2 교차 엔트로피 오차(cross entropy error, CEE)

$$E = -\sum_{k} t_k \log y_k$$

- log : 자연로그, y_k : 신경망의 출력, t_k : 정답 레이블(원-핫 인코딩)

그림 4-3 자연로그 $y = \log x$의 그래프



- 자연로그 그래프는 x가 1일 때 y는 0이 되고, x가 0에 가까울수록 y의 값은 점점 작아진다.
- 즉 그 출력이 1에 가까울수록 0에 다가가고, 반대로 정답일 때의 출력이 작아질수록 오차느 커진다.

```
# 크로스 엔트로피 손실 함수
def cross_entropy_error(y, t):
  delta = 1e-7 # 마이너스 무한대 수렴 방지 상수
  return -np.sum(t * np.log(y + delta))

y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
test1 = cross_entropy_error(np.array(y), np.array(t))
print(test1)

y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
test2 = cross_entropy_error(np.array(y), np.array(t))
print(test2)
```

4.2.3 미니배치 학습

- 모든 데이터에 대한 크로스 엔트로피 손실함수의 합 공식

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk}$$

- 단순히 1개의 데이터에서 N개의 데이터 합으로 푠현된것이다.
- 하지만 이 모든 데이터의 손실 함수를 구하는 방법은 매우 비효율적이고 시간이 많이 소모된다. → 따라서 미니배치 학습을 이용한다.
- **미니배치** : 훈련 데이터로부터 일부만 고른 것

- EX) 60,000장의 훈련 데이터 중에서 100장을 무작위로 뽑아 그 100장만을 사용하여 학습하는 방법 → **미니배치 학습**

```
#mnist 데이터 호출
import sys, os
sys.path.append("/content/drive/MyDrive/밑바닥부터 시작하는 딥러닝/code")
import numpy as np
from dataset.mnist import load_mnist

(x_train, t_train), (x_test, t_test) = load_mnist(normalize = True, one_hot_label = True)

print(x_train.shape) #(60000, 784)
print(t_train.shape) #(60000, 10)
#무작위로 10장 추출
train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]
```

4.2.4 (배치용) 교차 엔트로피 오차 구현하기

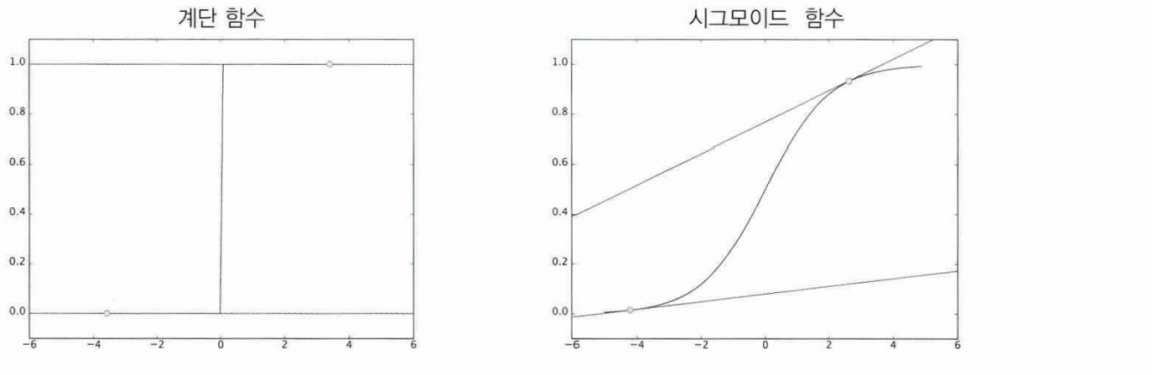
```
# y가 1차원, 데이터 하나당 reshape 함수로 데이터 형상 변경
# 배치의 크기로 나눠 정규화하고 이미지 1장당 평균의 교차 엔트로피 오차를 계산한다.
def cross_entropy_error(y,t):
  if y.ndim == 1:
    t = t.reshape(1, t.size)
    y = y.reshape(1, y.size)

  batch_size = y.shape[0]
  return -np.sum(t * np.log(y + 1e-7)) / batch_size

#정답 레이블이 원-핫 인코딩이 아니라 '2'나 '7'등의 숫자로 주어진 경우
#np.arange 함수를 이용하여 특정 함수값을 뽑아온다
def cross_entropy_error(y,t):
  if y.ndim == 1:
    t = t.reshape(1, t.size)
    y = y.reshape(1, y.size)

  batch_size = y.shape[0]
  return -np.sum(np.log(y[np.arange(batch_size), t] +  1e-7)) / batch_size
```

4.2.5 왜 손실 함수를 설정하는가?

- 높은 '정확도'를 위해서 정확도라는 지표를 놔두고 '손실 함수의 값'을 사용하는 이유는 미분의 역활에 있다.
- 가중치 매개변수의 손실 함수의 미분이란 '가중치 매개변수의 값을 아주 조금 변화 시켰을 때, 손실 함수가 어떻게 변하나'이다.
- 즉 신경망을 학습할때 정확도를 지표로 삼아서는 대부분의 미분 값이 0이 되어 매개변수의 갱신이 실패하기 떄문이다.
- 이 이유는 정확도가 지표라면 가중치 매개변수의 값을 조금 바꾼다하여도 정확도는 대부분 그대로 유지되고 이는 곧 미분 값이 0인 결과를 불러온다. 즉 정확도가 잘 개선되지 않고, 개선된다 할지라도, 연속적인 변화보다는 불연속적인 변화를 일으킨다.

  → 이는 활성화 함수로 계단 함수를 잘 사용하지 않는 이유와 비슷한데, 계단 함수의 미분이 대부분의 곳에서 0이기 떄문이다.(아래 그림 참고)

그림 4-4 계단 함수와 시그모이드 함수 : 계단 함수는 대부분의 장소에서 기울기가 0이지만, 시그모이드 함수의 기울기(접선)는 0이 아니다.



4.3 수치 미분(numerical diiferentiation)

4.3.1 미분

- 미분이란 한순간의 변화량을 표시한 것

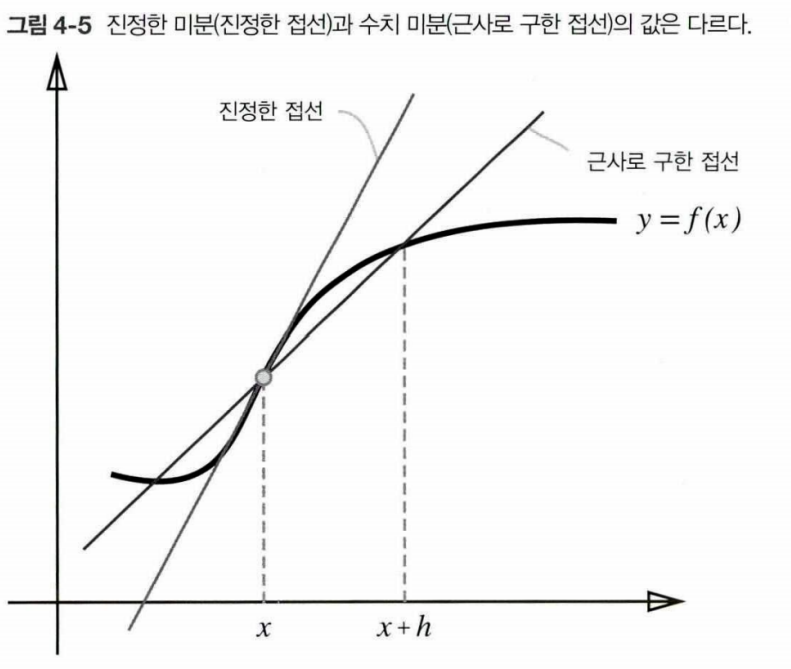
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```
# 나쁜 구현 예
def numerical_diff(f, x):
  h = 1e-50
  return (f(x+h) - f(x)) / h
```

- 해당 구현의 문제점 2가지

1. h의 값이 매우 작아 반올림 오차 문제를 일으킨다. → h를 e-4정도의 값으로 변경하자
2. f의 차분(임의 두점에서의 함수 값들의 차이)가 오차가 있다.즉 엄밀한 값이 아니다. 이 차분의 오류는 h를 무한히 0으로 좁히는 것이 불가능 → 중앙 차분(x + h, x - h)을 이용한다(아래 그림 참고)

그림 4-5 진정한 미분(진정한 접선)과 수치 미분(근사로 구한 접선)의 값은 다르다.

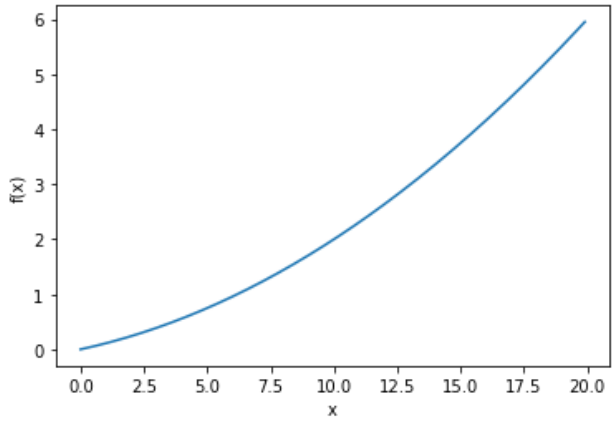



```
# 좋은 구현 예(수치 미분)

def numerical_diff(f, x):
  h = 1e-4 # 0.0001
  return (f(x+h) - f(x-h)) / (2 * h)
```

4.2.3 수치 미분의 예

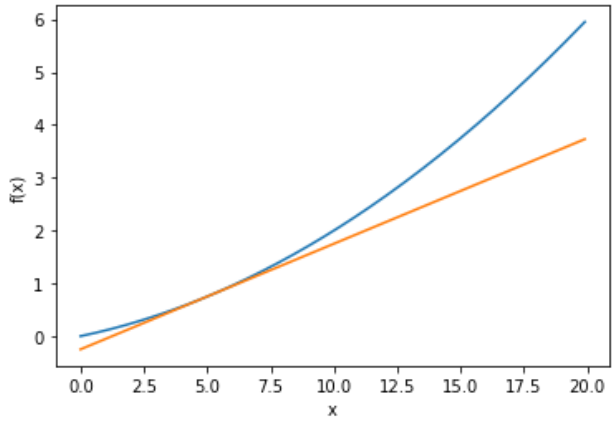
$$y = 0.01x^2 + 0.1x$$

```
def function_1(x):
  return 0.01* x **2 + 0.1 * x

import numpy as np
import matplotlib.pylab as plt

x = np.arange(0.0, 20.0, 0.1) #0에서 20까지 0.1간격의 배열 x를 만든다.(20은 미포함)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x, y)
plt.show()
```

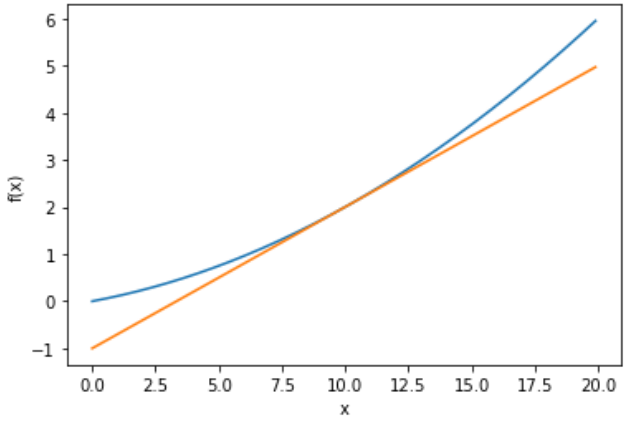



```
print(numerical_diff(function_1, 5)) # 0.1999999999990898
print(numerical_diff(function_1, 10)) #0.2999999999986347
```

- x=5 일 때 함수의 접선



- x=10 일 때, 함수의 접선



4.3.3 편미분

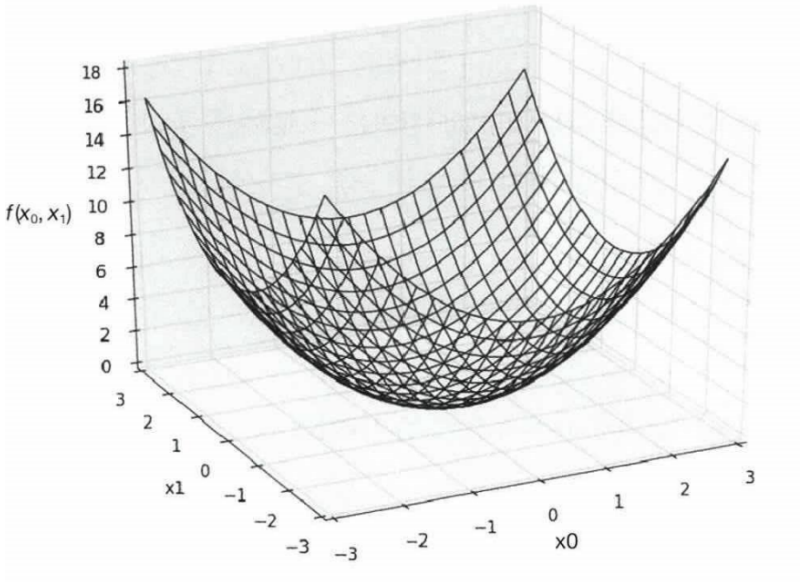
- 변수가 2개인 미분

$$f(x_0, x_1) = x_0^2 + x_1^2$$

```
def function_2(x):
  return x[0] ** 2 + x[1] ** 2
  #또는 return np.sum(x**2)
```

- 코드는 생각보다 간단하다, 이를 3차원 그래프로 그려보자

그림 4-8 $f(x_0, x_1) = x_0^2 + x_1^2$의 그래프




```
#x_0에 대한 편미분
def function_tmp1(x0):
  return x0 * x0 + 4.0 ** 2.0

print(numerical_diff(function_tmp1, 3.0)) #6.00000000000378

#x_1에 대한 편미분
def function_tmp2(x1):
  return 3.0 ** 2.0 + x1 * x1

print(numerical_diff(function_tmp2, 4.0)) #7.999999999999119
```

- 이처럼 편미분은 여러 변수 중 목표 변수 하나에 초점을 맞추고 다른 변수는 값을 고정한다.

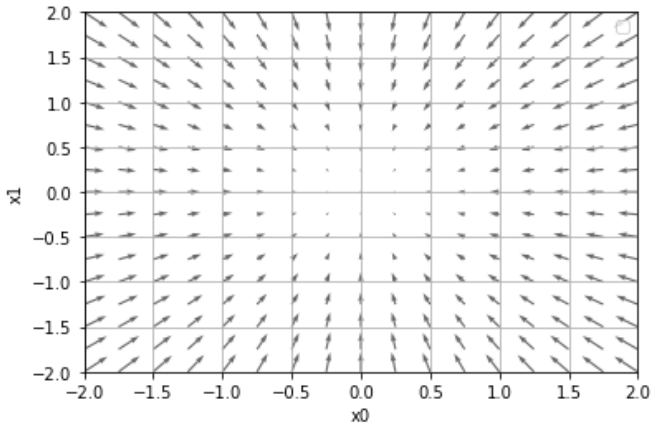
4.4 기울기

- 모든 변수의 편미분을 벡터로 정리한 것

```
def numerical_gradient(f,x):
  h = 1e-4
  grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성

  for idx in range(x.size):
    tmp_val = x[idx]
```

```
    # f(x + h) 계산
    x[idx] = tmp_val + h
    fxh1 = f(x)

    #f(x-h) 계산
    x[idx] = tmp_val - h
    fxh2 = f(x)

    grad[idx] = (fxh1 - fxh2) / (2*h)
    x[idx] = tmp_val # 값 복원

  return grad

# (3,4), (0,2), (3,0)에서의 기울기 계산

print(numerical_gradient(function_2, np.array([3.0, 4.0]))) # [6. 8.]
print(numerical_gradient(function_2, np.array([0.0, 2.0]))) # [0. 4.]
print(numerical_gradient(function_2, np.array([3.0, 0.0]))) # [6. 0.]
```

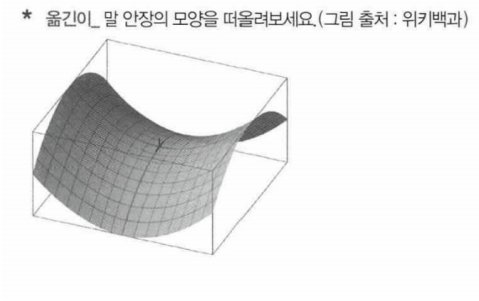
- 기울기 : 함수의 '가장 낮은 장소(최솟값)를 가리킨다.→ 반드시는 아니고 실제로 기울기는 각 지점에서 낮아지는 방향을 가리킨다.



- 즉 **기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향**이다

4.4.1 경사법(경사 하강법)

- 경사법 : 기울기를 잘 이용해 함수의 최솟값 즉 손실함수가 최솟값이 될 때의 매개변수를 찾는 방법
- 주의 사항 : 함수가 극솟값, 최솟값 또 안장점(saddle point)에서 기울기가 0이다. 경사법은 이러한 기울기가 0인 곳을 찾지만 그 값이 항상 최솟값인(global minimum)인 것은 아니다. 또한 특정 모양의 평평한 곳인 고원(plateau)에 빠질수도 있다.



- 경사 하강법 : 최솟값 탐색, 경사 상승법 : 최댓값 탐색

- 경사법 수식

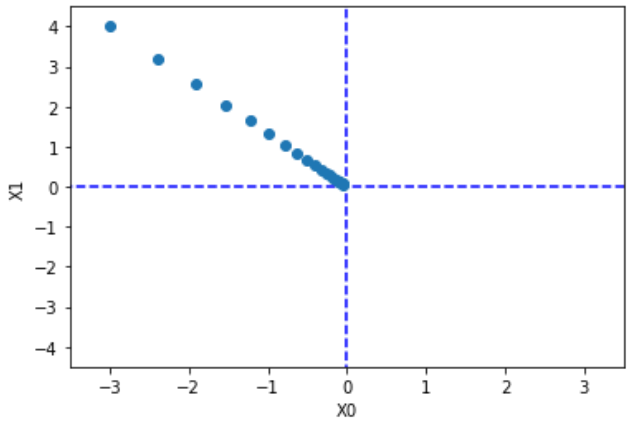
$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

- 위의 수식을 여러번 적용하고 학습률은 적절히 정해주어야 한다.
- 학습률 : 한 번의 학습으로 매개변수 값을 얼마나 갱신할것인지 정하는 지표

```
def gradient_descent(f, init_x, lr=0.01, step_num = 100):
  x = init_x

  for i in range(step_num):
    grad = numerical_gradient(f, x)
    x -= lr * grad

  return x
```

```
def function_2(x):
  return x[0] ** 2 + x[1] ** 2

init_x = np.array([-3.0, 4.0])
grad_desc = gradient_descent(function_2, init_x = init_x, lr =0.1, step_num = 100)
print(grad_desc) #[-6.11110793e-10  8.14814391e-10]
```



4.4.2 신경망에서의 기울기

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient

#간단한 신경망 구현
class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3) # 정규분포로 초기화

    def predict(self, x):
        return np.dot(x, self.W)

    def loss(self, x, t):
        z = self.predict(x)
        y = softmax(z)
        loss = cross_entropy_error(y, t)

        return loss

x = np.array([0.6, 0.9]) #입력 데이터
t = np.array([0, 0, 1]) # 정답 레이블

net = simpleNet()
print(net.W)
#[[ 0.25306196 -1.15445558 -0.38891805]
# [-0.46583819  0.31229111 -0.08818537]]
#기울기 구현
f = lambda w: net.loss(x, t)
dW = numerical_gradient(f, net.W)

print(dW)
#[[ 0.21265806  0.18410226 -0.39676033]
# [ 0.31898709  0.2761534  -0.59514049]]
```

4.5 학습 알고리즘 구현하기

- 전제 : 신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 조정하는 과정을 '학습'이라 한다.

1단계 - 미니배치

- 훈련 데이터 중 일부를 무작위로 가져온다.

2단계 - 기울기 산출

- 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다.

3단계 - 매개변수 갱신

- 가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4단계 - 반복

- 1 ~ 3단계를 반복한다.

- 확률적 경사 하강법(Stochastic gradient descent) : 확률적으로 무작위로 골라낸 데이터에 대해 수행하는 경사 하강법

4.5.1 2층 신경망 클래스 구현하기

- 2층 신경망을 하나의 클래스로 구현한다.

```
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
from common.functions import *
from common.gradient import numerical_gradient


class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x : 입력 데이터, t : 정답 레이블
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x : 입력 데이터, t : 정답 레이블
```

```
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])

        return grads

    def gradient(self, x, t):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        grads = {}

        batch_num = x.shape[0]

        # forward
        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        # backward
        dy = (y - t) / batch_num
        grads['W2'] = np.dot(z1.T, dy)
        grads['b2'] = np.sum(dy, axis=0)

        da1 = np.dot(dy, W2.T)
        dz1 = sigmoid_grad(a1) * da1
        grads['W1'] = np.dot(x.T, dz1)
        grads['b1'] = np.sum(dz1, axis=0)

        return grads
```

- 겹치는 내용이 많아 설명은 생략한다.

표 4-1 TwoLayerNet 클래스가 사용하는 변수

변수	설명
params	신경망의 매개변수를 보관하는 딕셔너리 변수(인스턴스 변수)
	params['W1']은 1번째 층의 가중치, params['b1']은 1번째 층의 편향
	params['W2']는 2번째 층의 가중치, params['b2']는 2번째 층의 편향
grads	기울기 보관하는 딕셔너리 변수(numerical_gradient() 메서드의 반환 값)
	grads['W1']은 1번째 층의 가중치의 기울기, grads['b1']은 1번째 층의 편향의 기울기
	grads['W2']는 2번째 층의 가중치의 기울기, grads['b2']는 2번째 층의 편향의 기울기

표 4-2 TwoLayerNet 클래스의 메서드

메서드	설명
__init__(self, input_size, hidden_size, output_size)	초기화를 수행한다. 인수는 순서대로 입력층의 뉴런 수, 은닉층의 뉴런 수, 출력층의 뉴런 수
predict(self, x)	예측(추론)을 수행한다. 인수 x는 이미지 데이터
loss(self, x, t)	손실 함수의 값을 구한다. 인수 x는 이미지 데이터, t는 정답 레이블(아래 칸의 세 메서드의 인수들도 마찬가지)
accuracy(self, x, t)	정확도를 구한다.
numerical_gradient(self, x, t)	가중치 매개변수의 기울기를 구한다.
gradient(self, x, t)	가중치 매개변수의 기울기를 구한다. numerical_gradient()의 성능 개선판! 구현은 다음 장에서...

- 해당 코드로 실행 가능

```
x = np.random.rand(100, 784)
y = net.predict(x)
```

4.5.2 미니배치 학습 구현하기

```
import sys, os
sys.path.append("/content/drive/MyDrive/밑바닥부터 시작하는 딥러닝/MyCode")  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

train_loss_list = []

# 하이퍼파라미터
iters_num = 10000  # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100   # 미니배치 크기
learning_rate = 0.1

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    grad = network.numerical_gradient(x_batch, t_batch)
    #grad = network.gradient(x_batch, t_batch)
```

```
    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
```

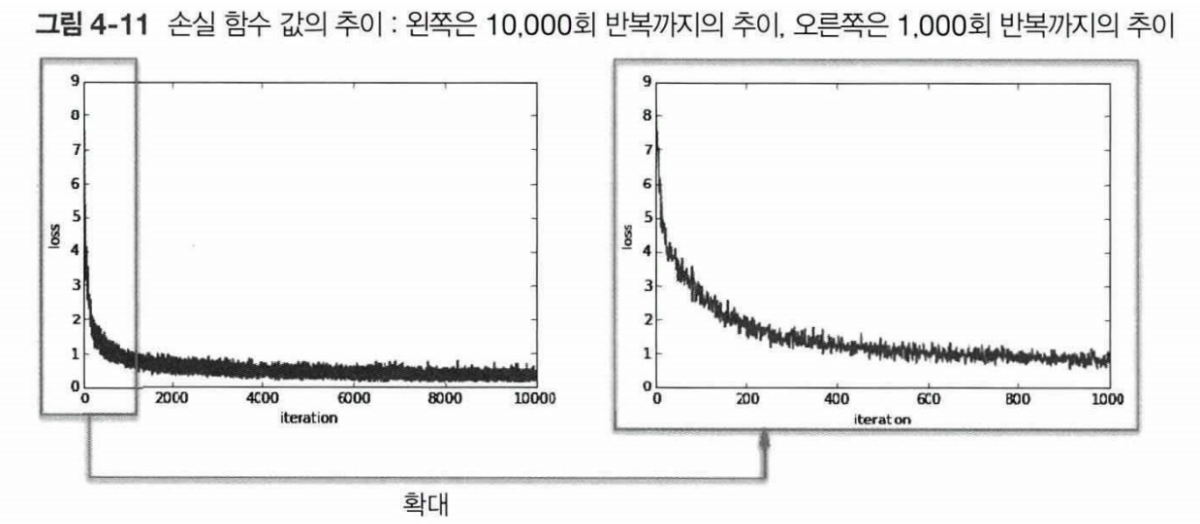


그림 4-11 손실 함수 값의 추이 : 왼쪽은 10,000회 반복까지의 추이, 오른쪽은 1,000회 반복까지의 추이

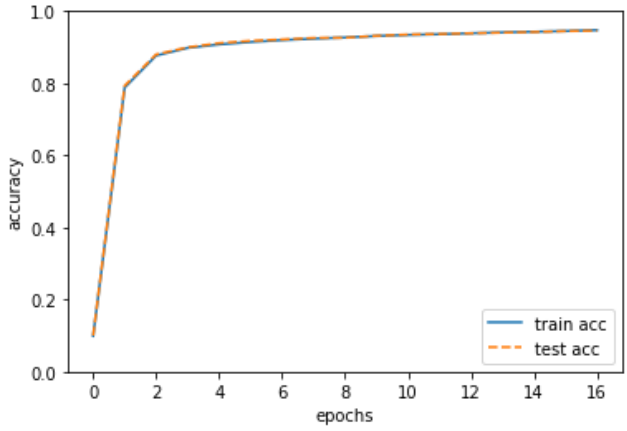
4.5.3 시험 데이터로 평가하기(MNIST 데이터)

- 기울기 계산의 편의성을 위해 numerical_gradient 대신에 gradient를 사용함

→ 실제로 시간의 차이를 체감함(다음장에서 구현함)

```
# coding: utf-8
import sys, os
sys.path.append("/content/drive/MyDrive/밑바닥부터 시작하는 딥러닝/MyCode")  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

# 하이퍼파라미터
iters_num = 10000  # 반복 횟수를 적절히 설정한다.
train_size = x_train.shape[0]
batch_size = 100   # 미니배치 크기
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

# 1에폭당 반복 수
iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    # 미니배치 획득
    batch_mask = np.random.choice(train_size, batch_size)
```

```
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 기울기 계산
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)

    # 매개변수 갱신
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    # 학습 경과 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    # 1에폭당 정확도 계산
    if i % iter_per_epoch == 0:
        train_acc = network.accuracy(x_train, t_train)
        test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# 그래프 그리기
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```



- 에폭이 진행될수록 훈련 데이터와 시험 데이터를 평가한 정확도가 모두 좋아지고 있다. 즉 오버피팅이 일어나지 않고 있다!