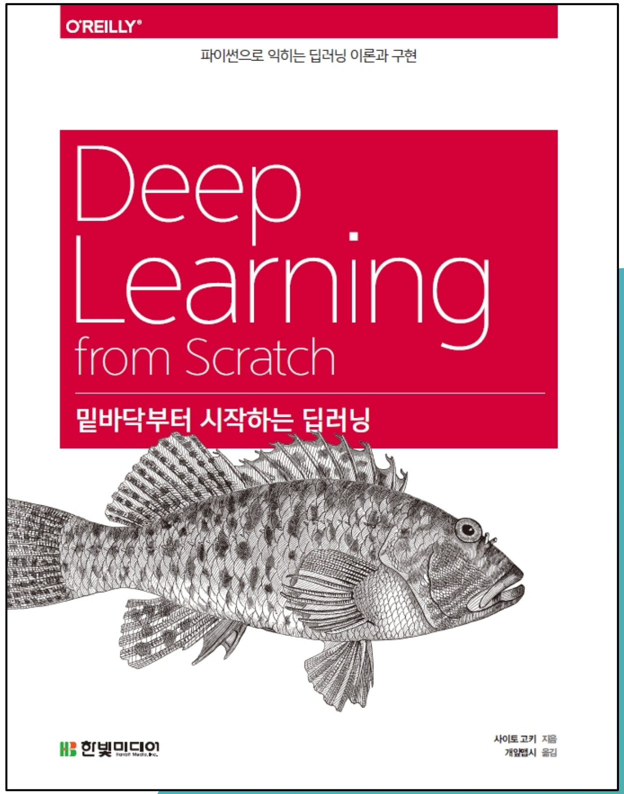
CHAPTER 4 신경망 학습

밑바닥부터 시작하는 딥러닝



홍익대학교 컴퓨터공학과
박 준

이 책의 학습 목표

- CHAPTER 1 파이썬에 대해 간략하게 살펴보고 사용법 익히기
- CHAPTER 2 퍼셉트론에 대해 알아보고 퍼셉트론을 써서 간단한 문제를 풀어보기
- CHAPTER 3 신경망의 개요, 입력 데이터가 무엇인지 신경망이 식별하는 처리 과정 알아보기
- CHAPTER 4 손실 함수의 값을 가급적 작게 만드는 경사법에 대해 알아보기
- CHAPTER 5 가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법 배우기
- CHAPTER 6 신경망(딥러닝) 학습의 효율과 정확도를 높이기
- CHAPTER 7 CNN의 메커니즘을 자세히 설명하고 파이썬으로 구현하기
- CHAPTER 8 딥러닝의 특징과 과제, 가능성, 오늘날의 첨단 딥러닝에 대해 알아보기

Contents

Contents

○ CHAPTER 4 신경망 학습

CHAPTER 4 신경망 학습

손실 함수의 값을 가급적 작게 만드는 경사법에 대해 알아보기

4.1.1 데이터 주도 학습

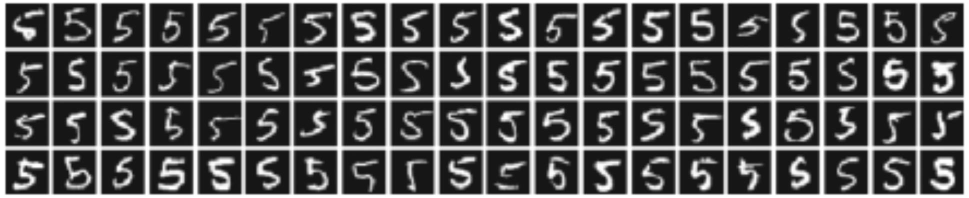


그림 4-1 손글씨 숫자 ‘ 5 ’의 예 : 사람마다 자신만의 필체가 있다

기계학습은 데이터가 생명.
그래서 기계학습의 중심에는 데이터가.
이처럼 데이터가 이끄는 접근 방식 덕에 사람 중심 접근에서 벗어날 수 있다.

그림 4-2 규칙을 ‘사람’이 만드는 방식에서 ‘기계’가 데이터로부터 배우는 방식으로의 패러다임 전환 : 회색 블록은 사람

[그림 4 - 2]와 같이 신경망은 이미지를 ‘있는 그대로’ 학습한다.
두 번째 접근 방식(특징과기계학습 방식)에서는 특징을 사람이 설계했지만, 신경망은 이미지에 포함된 중요한 특징까지도 ‘기계’가 스스로 학습할 것이다.

4.1.1 데이터 주도 학습

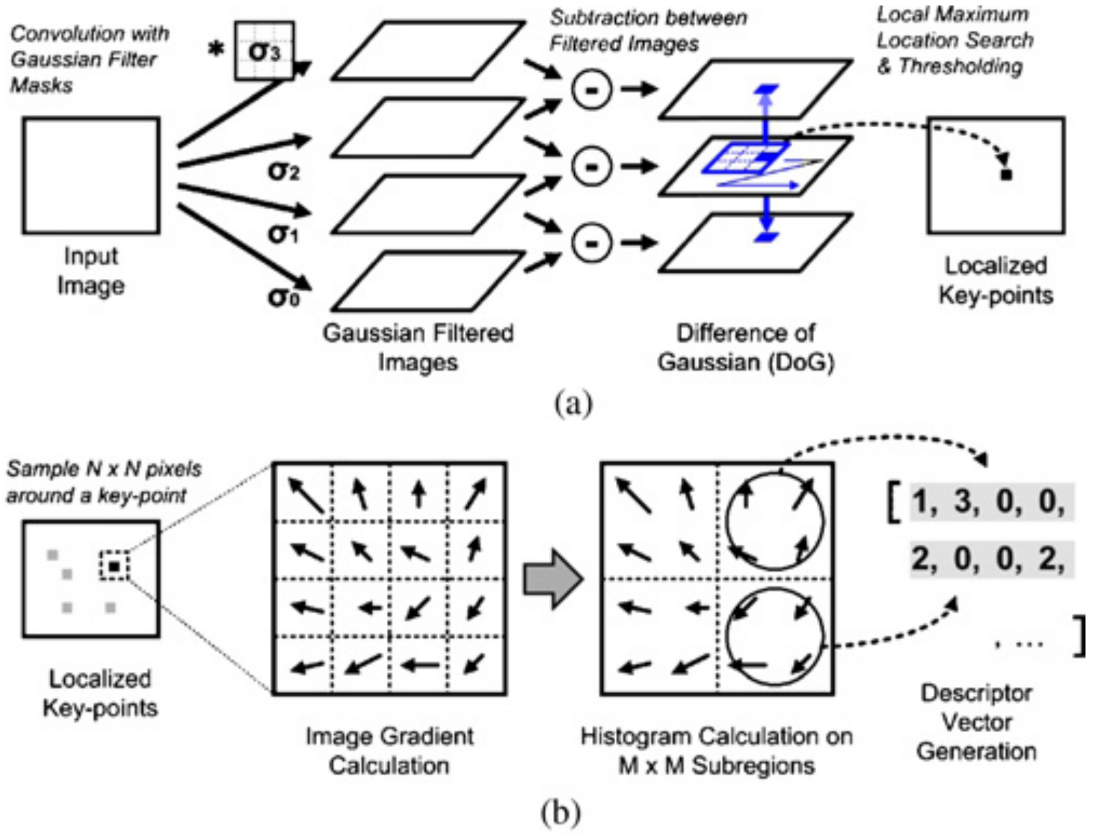
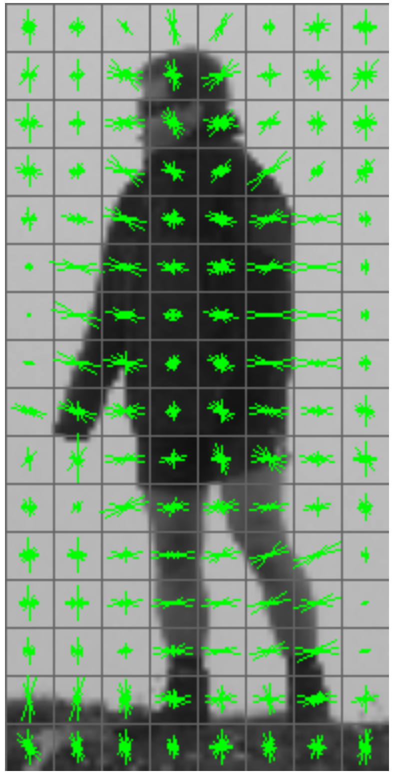


	Image Filtering	DoG	LMLS	Descriptor Gen.
Computation Load	52%	5%	27%	16%



4.1.2 훈련 데이터와 시험 데이터

기계학습 문제는 데이터를 훈련 데이터 training data 와 시험 데이터 test data 로 나누어 학습과 실험을 수행하는 것이 일반적.

- 훈련 데이터만 사용하여 학습하면서 최적의 매개변수를 찾음
- 시험 데이터를 사용하여 앞서 훈련한 모델의 성능을 평가

Why?

우리가 원하는 것은 범용적으로 사용할 수 있는 모델이기 때문. 이 **범용 능력** 을 제대로 평가하기 위해 훈련 데이터와 시험 데이터를 분리하는 것.

- 데이터셋 하나로만 매개변수의 학습과 평가를 수행하면 올바른 평가가 될 수 없다.
- 수중의 데이터셋은 제대로 맞히더라도 다른 데이터셋에는 엉망인 일도 벌어진다.
- 하나의 데이터셋에만 지나치게 최적화된 상태를 오버피팅 overfitting * 이라고 한다.
- 오버피팅 피하기는 기계학습의 중요한 과제

4.2 손실 함수

신경망 학습에서는 현재의 상태를 '하나의 지표'로 표현
그리고 그 지표를 가장 좋게 만들어주는 가중치 매개변수의 값을 탐색하는 것.

신경망 학습에서 사용하는 지표는
손실 함수 loss function / cost function / error function / objective function

이 손실 함수는 임의의 함수를 사용할 수도 있지만 일반적으로는 평균 제곱 오차와 교차 엔트로피 오차를 사용.

4.2.1 평균 제곱 오차

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2 \qquad \text{[식 4.1]}$$

여기서 y_k 는 신경망의 출력(신경망이 추정한 값)
t_k 는 정답 레이블
k 는 데이터의 차원 수

이를테면 “MNIST” 예에서
y_k 와 t_k 는 다음과 같은 원소 10 개짜리 데이터이다.

```
>>> 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]
```

이 배열들의 원소는 첫 번째 인덱스부터 순서대로 숫자 ‘ 0 ’, ‘ 1 ’, ‘ 2 ’, ...일 때의 값.
여기에서 신경망의 출력 y 는 소프트맥스 함수의 출력

이처럼 한 원소만 1 로 하고 그 외는 0 으로 나타내는 표기법: 원-핫 인코딩

4.2.2 교차 엔트로피 오차

또 다른 손실 함수로서 교차 엔트로피 오차 cross entropy error , CEE 도 자주 이용한다.
교차 엔트로피 오차의 수식은 다음과 같다.

$$E = -\sum_k t_k \log y_k \qquad \text{[식 4.2]}$$

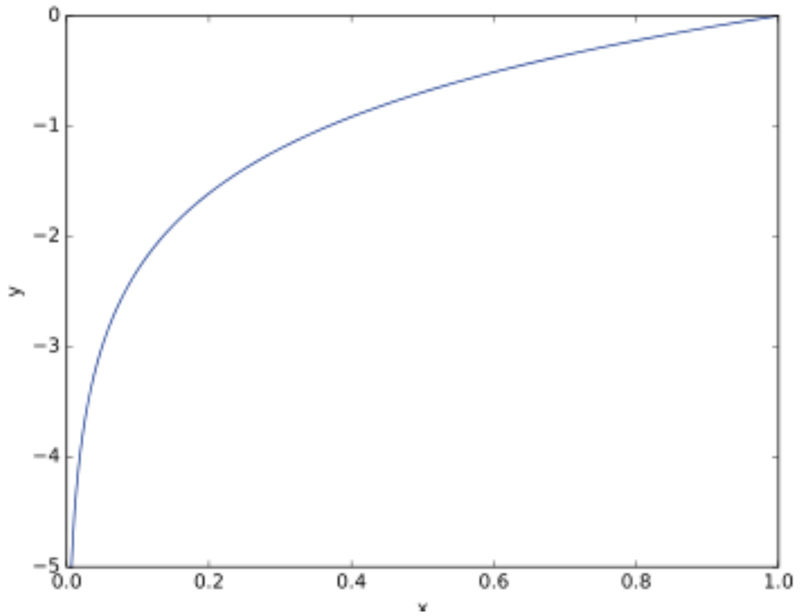


그림 4-3 자연로그 y = logx 의 그래프

x 가 1 일 때 y 는 0 이 되고 x 가 0 에 가까워질수록 y 의 값은 점점 작아진다.
[식 4 . 2]도 마찬가지로 정답에 해당하는 출력이 커질수록 0 에 다가가다가, 그 출력이 1 일 때 0이 된다.
반대로 정답일 때의 출력이 작아질수록 오차는 커진다.

4.2.3 미니배치 학습

기계학습 문제: 훈련 데이터에 대한 손실 함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아낸다.

이렇게 하려면 모든 훈련 데이터를 대상으로 손실 함수 값을 구해야 한다.
즉, 훈련 데이터가 100 개 있으면 그로부터 계산한 100 개의 손실 함수 값들의 합을 지표로 삼는 것.

$$E = -\frac{1}{N}\sum_{n}\sum_{k} t_{nk}\log y_{nk} \qquad [식\ 4.3]$$

이때 데이터가 N 개라면 t_{nk} 는 n 번째 데이터의 k 번째 값을 의미한다
(y_{nk} 는 신경망의 출력, t_{nk}는 정답 레이블).
수식이 좀 복잡해 보이지만 데이터 하나에 대한 손실 함수인 [식 4 . 2]를 단순히 N 개의 데이터로 확장했을 뿐.

신경망 학습에서도 훈련 데이터로부터 일부(mini-batch)만 골라 학습을 수행합니다.

예) 60,000 장의 훈련 데이터 중에서 100 장을 무작위로 뽑아그 100 장만을 사용하여 학습하는 것.
:미니배치 학습

4.2.3 미니배치 학습

```
train_size = x.shape[0]
batch_size = min(train_size, 100)

batch_mask = np.random.choice(train_size, batch_size)
x_batch = x[batch_mask]
t_batch = t[batch_mask]
```

For MNIST

```
print (np.random.choice(60000, 10))
```

4.2.4 (배치용) 교차 엔트로피 오차 구현하기

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
```

y 가 1 차원이라면, 즉 데이터 하나당 교차 엔트로피 오차를 구하는 경우는 reshape 함수로 데이터의 형상을 바꿔준다 (mini-batch로 일반화 시키기 위해)

그리고 배치의 크기로 나눠 이미지 1 장당 평균의 교차 엔트로피 오차를 계산.

실 습

```
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
```

이 구현에서는 원-핫 인코딩일 때 t 가 0 인 원소는 교차 엔트로피 오차도 0 이므로, 그 계산은무시해도 좋다는 것이 핵심이다.
다시 말하면 정답에 해당하는 신경망의 출력만으로 교차 엔트로피 오차를 계산할 수 있다.
그래서 원-핫 인코딩 시 t * np . log (y)였던 부분을 레이블 표현일 때는 np . log (y [np . arange (batch _ size), t])로 구현한다

4.2.4 (배치용) 교차 엔트로피 오차 구현하기

```
y1 = np.array([0.1, 0.05, 0, 0.6, 0, 0.1, 0, 0.4, 0.05, 0])
t1 = np.array([0, 0, 0, 1, 0, 0, 0, 0, 0, 0])



y2 = np.array([[0.1, 0.05, 0, 0.6, 0, 0.1, 0, 0.4, 0.05, 0],
               [0.1, 0.05, 0, 0.06, 0, 0.1, 0, 0.4, 0.5, 0]])
t2 = np.array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]])


print (cross_entropy_error(y1, t1))
print (cross_entropy_error(y2, t2))
```

4.2.5 왜 (정확도를 사용하지 않고) 손실 함수를 설정하는가?

정확도는 매개변수의 미소한 변화에는 거의 반응을 보이지 않고, 반응이 있더라도 그 값이 불연속적으로 갑자기 변화함.

이는 '계단 함수'를 활성화 함수로 사용하지 않는 것과 유사한 이유

- 만약 활성화 함수로 계단 함수를 사용하면 신경망 학습이 잘 이뤄지지 않음
- 계단 함수의 미분은 [그림 4 – 4]와 같이 대부분의 장소(0이외의 곳)에서 0
- 그 결과, 계단 함수를 이용하면 손실 함수를 지표로 삼는 게 아무 의미가 없게 된다.
- 매개변수의 작은 변화가 주는 파장을 계단 함수가 말살하여 손실 함수의 값에는 아무런 변화가 나타나지 않기 때문

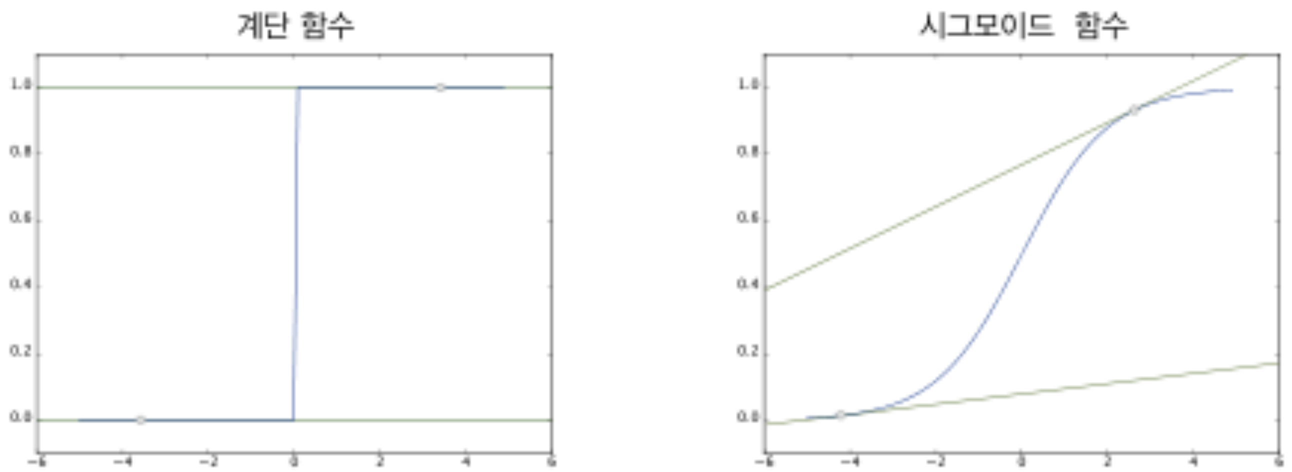
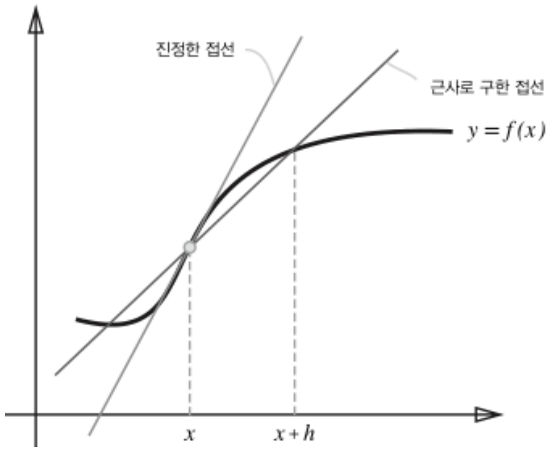


그림 4-4 계단 함수와 시그모이드 함수 : 계단 함수는 대부분의 장소에서 기울기가 0 이지만, 시그모이드 함수의 기울기(접선)는 0 이 아니다

4.3.1 미분

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$ [식 4.4]

[식 4 . 4]는 함수의 미분을 나타낸 식이다 (numerical difference)
좌변은 f (x)의 x 에 대한 미분(x 에 대한 f (x)의 변화량)을 나타내는 기호이다.
결국, x 의 '작은 변화'가 함수 f (x)를 얼마나 변화시키느냐를 의미
이때 시간의 작은 변화, 즉 시간을 뜻하는 h 를 한없이 0 에 가깝게 한다는 의미



[그림 4 - 5]와 같이 수치 미분에는 오차가 포함된다. 이 오차를 줄이기 위해 (x + h)와 (x -h)일 때의 함수 f 의 차분을 계산하는 방법을 쓰기도 한다.
이 차분은 x 를 중심으로 그 전후의차분을 계산한다는 의미에서 중심 차분 혹은 중앙 차분 이라 한다
(한편,(x + h)와 x 의 차분은 전방 차분 이라 함).

numerical_difference(f, x) 구현

4.3.2 수치 미분의 예

$$y = 0.01x^2 + 0.1x$$ [식 4.5]

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

[식 4 . 5]를 파이썬으로 구현

```
import numpy as np
import matplotlib.pylab as plt

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

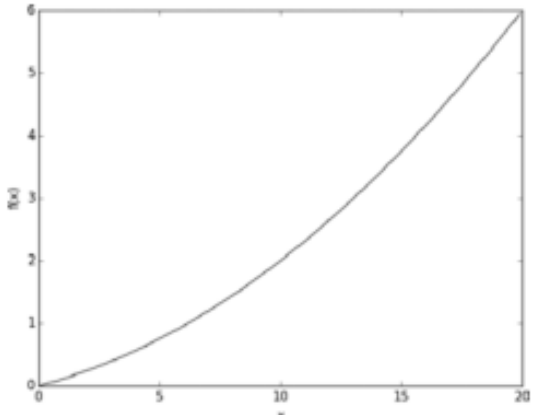


그림 4-6 식
f (x) = 0 . 01x 2 + 0 . 1x 의 그래프

```
>>> numerical_diff(function_1, 5)
0.1999999999990898
>>> numerical_diff(function_1, 10)
0.2999999999986347
```

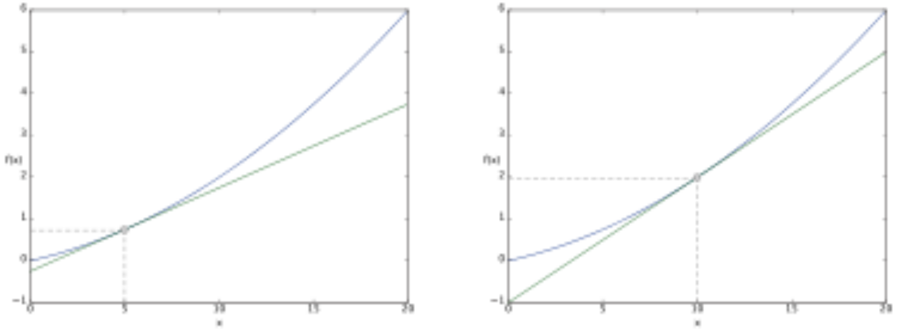


그림 4-7 x = 5 , x = 10 에서의 접선 : 직선의 기울기는 수치 미분에서 구한 값을 사용하였다.

4.3.2 수치 미분의 예

numerical_difference(f, x) 구현

```python
def f1(x):
    return (0.01*x*x+0.1*x)

print(numerical_difference(f1, 5))
print(numerical_difference(f1, 10))
```

```
>>> numerical_diff(function_1, 5)
0.1999999999990898
>>> numerical_diff(function_1, 10)
0.2999999999986347
```

4.3.3 편미분

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

[식 4.6]

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

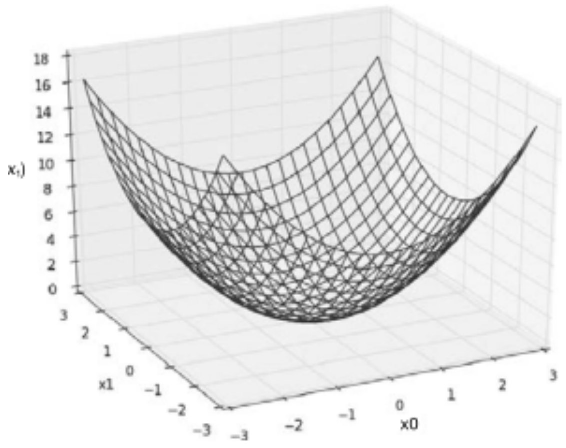


그림 4-8 f (x 0 , x 1) = x 0 2 + x 1 2 의 그래프

즉 x 0 와 x 1 중 어느 변수에 대한 미분이냐를 구별해야 한다.
이와 같이 변수가 여럿인 함수에 대한 미분을 편미분 이라고 한다.

4.4 기울기

```
def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열을 생성
```

실제로는 [6 . 0000000000037801 , 7 . 9999999999991189]라는 값이 얻어지지만 [6 ., 8 .]으로 출력된다.
이는 넘파이 배열을 출력할 때 수치를 '보기 쉽도록' 가공하기 때문이다

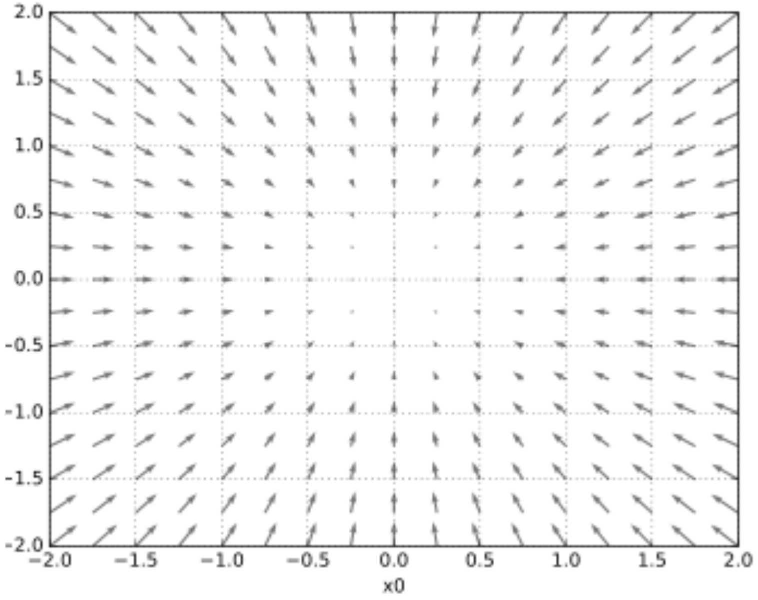


그림 4-9 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 기울기

기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향

4.4 기울기

```
def f2(x):
    return x[0]*x[0]+x[1]*x[1]

x = np.array([3.0, 4.0])
print(numerical_gradient1(f2, x))
```

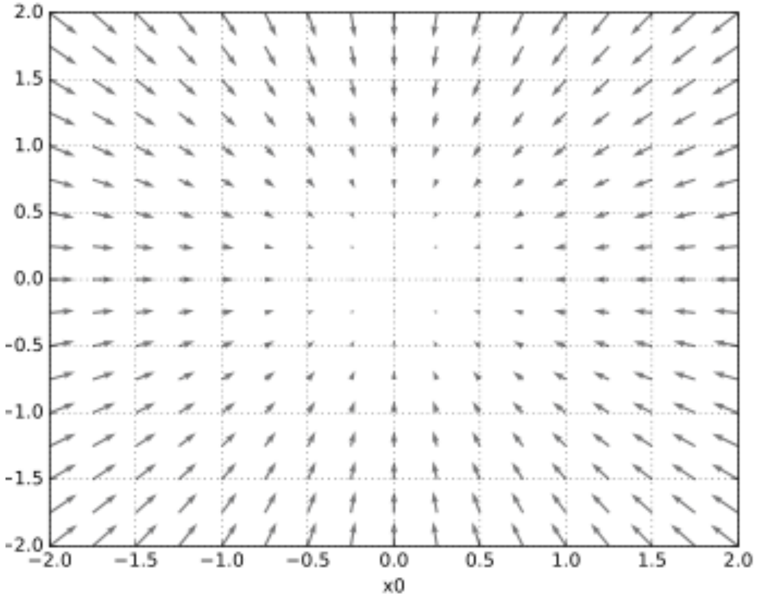


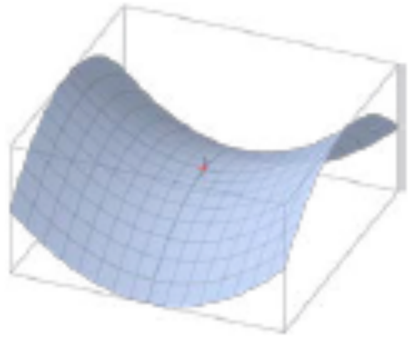
그림 4-9 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 기울기

기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향

실제로는 [6 . 0000000000037801 , 7 . 9999999999991189]라는 값이 얻어지지만 [6 ., 8 .]으로 출력된다.
이는 넘파이 배열을 출력할 때 수치를 '보기 쉽도록' 가공하기 때문이다

4.4.1 경사법(경사 하강법): Gradient Descent

매개변수 공간이 광대하여 어디가 최솟값이 되는 곳인지를 짐작할 수 없다. 이런 상황에서 기울기를잘 이용해 함수의 최솟값(또는 가능한 한 작은 값)을 찾으려는 것이 경사법이다.



문제 : 경사법으로 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 최솟값을 구하라

```
>>> def function_2(x):
...     return x[0]**2 + x[1]**2
...
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
array([ -6.11110793e-10,   8.14814391e-10])
```

gradient_descent(f, init_x, lr=0.1, step_num=100) 구현

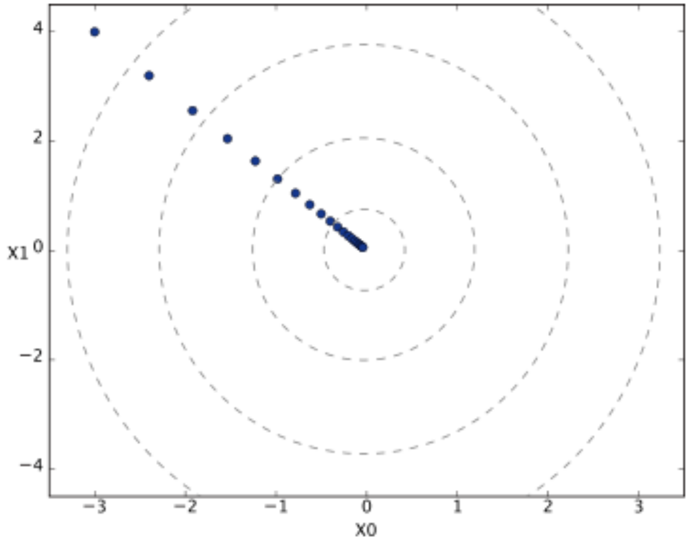


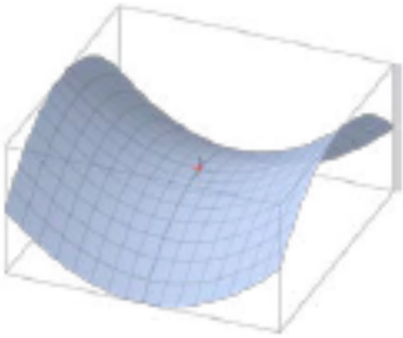
그림 4-10 경사법에 의한 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 갱신 과정 : 점선은 함수의 등고선을 나타낸다

4.4.1 경사법(경사 하강법): Gradient Descent

문제 : 경사법으로 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 최솟값을 구하라

```
>>> def function_2(x):
...     return x[0]**2 + x[1]**2
...
>>> init_x = np.array([-3.0, 4.0])
>>> gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100)
array([ -6.11110793e-10,   8.14814391e-10])
```

Test
gradient_descent(f2, init_x, lr=0.1, step_num=100)
gradient_descent(f2, init_x, lr=10.0, step_num=100)
gradient_descent(f2, init_x, lr=1e-5, step_num=100)



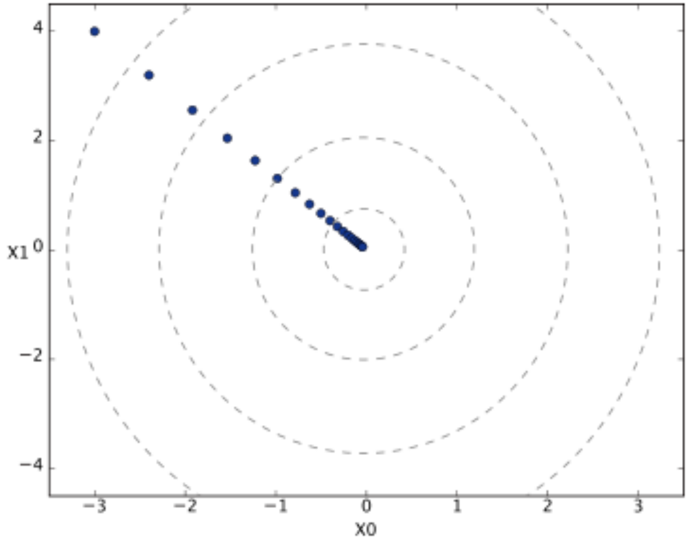


그림 4-10 경사법에 의한 $f(x_0, x_1) = x_0^2 + x_1^2$ 의 갱신 과정 : 점선은 함수의 등고선을 나타낸다

4.4.2 신경망에서의 기울기

[식 4.8]

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

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

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from common.functions import softmax, cross_entropy_error
from common.gradient import numerical_gradient

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
```

신경망 학습에서도 기울기를 구해야 한다.

여기서 말하는 기울기는 가중치 매개변수에 대한
손실 함수의 기울기입니다

이전에 구현한 softmax 와 cross_entropy_error 메서드를 이용.

그리고 numerical_ gradient 메서드도 이용.

4.5 학습 알고리즘 구현하기

전제

신경망에는 적응 가능한 가중치와 편향이 있고,
이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다.
신경망 학습은 다음과 같이 4 단계로 수행한다.

1 단계 - 미니배치

훈련 데이터 중 일부를 무작위로 가져온다.
이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실함수 값을 줄이는 것이 목표.

2 단계 - 기울기 산출

미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구한다. 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시한다.

3 단계 - 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4 단계 - 반복

1 ~ 3 단계를 반복한다.

4.5.1 2층 신경망 클래스 구현하기

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 4-1 TwoLayerNet 클래스가 사용하는 변수

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...

표 4-2 TwoLayerNet 클래스의 메서드

4.5.1 2층 신경망 클래스 구현하기

```
import sys, os
sys.path.append(os.pardir)
from common.functions import *
from common.gradient import numerical_gradient

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size,

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

TwoLayerNet 의 구현은 스탠퍼드 대학교의 CS231n 수업에서
제공한 파이썬 소스 코드를 참고.

```
# 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):

# x : 입력 데이터, t : 정답 레이블
def numerical_gradient(self, x, t):
```

4.5 학습 알고리즘 구현하기

Stochastic Gradient Descent (SGD, 확률적 경사 하강법)
데이터를 미니배치로 무작위 선정

```
def learn(self, lr = 0.01, epoch = 100, batch_size = 1, verbose = True):
    """pre-requisite: x, t are stored in the local attribute"""
    batch_size = max(batch_size, self.x.shape[0])

    for k in range(epoch):
        if verbose:
            print("cost, accuracy:", \
                  self.loss(self.x, self.t), self.accuracy(self.x, self.t))
        batch_mask = np.random.choice(self.x.shape[0], batch_size)
        x_batch = self.x[batch_mask]
        t_batch = self.t[batch_mask]
```

4.5 학습 알고리즘 구현하기

Stochastic Gradient Descent (SGD, 확률적 경사 하강법)
데이터를 미니배치로 무작위 선정

```
tlnn = Two_Layer_Neural_Network(num_input_layer, num_hidden_layer, num_output_layer)
tlnn.init_data(x_train, t_train)

tlnn.learn(lr=0.01, epoch=10000, batch_size=40)

tr = tlnn.accuracy(x_train, t_train)
te = tlnn.accuracy(x_test, t_test)
print(tr)
print(te)
```

4.5 학습 알고리즘 구현하기

Numerical Gradient
- need to extend to 2 dimension

```
def numerical_gradient(f, X):
    if X.ndim == 1:
        return _numerical_gradient_no_batch_(f, X)   # original numerical_gradient function
    else:
        grad = np.zeros_like(X)

        for idx, x in enumerate(X):
            # print("in numerical gradient ", idx, x)
            grad[idx] = _numerical_gradient_no_batch_(f, x)

        return grad
```