Chapter 4. 신경망 학습

- 여기서의 **학습** 이란 훈련 데이터로부터 가중치 매개변수의 최적값을 자동으로 획득하는 것
- 신경망이 학습할 수 있도록 해주는 **지표**인 손실 함수
- 이 손실 함수의 결괏값을 가장 작게 만드는 가중치 매개변수를 찾는 것이 학습의 목표
- 손실 함수의 값을 가급적 작게 만드는 기법인 함수의 기울기를 활용한 **경사법**
-

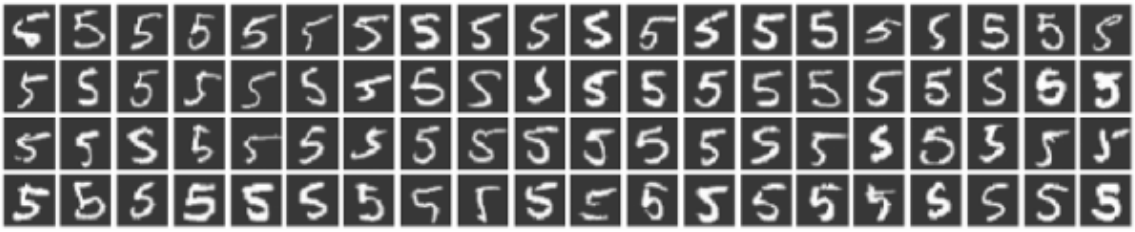
4-1. 데이터에서 학습

- 신경망의 특징은 데이터를 보고 학습할 수 있다는 점
- 데이터에서 학습한다는 것은 가중치 매개변수의 값을 데이터를 보고 자동으로 결정한다는 뜻
- 퍼셉트론도 직선으로 분리할 수 있는(선형 분리 가능) 문제라면, 데이터로부터 자동으로 학습 가능함
- 선형 분리 가능 문제는 유한 번의 학습을 통해 풀 수 있다는 사실이 **퍼셉트론 수렴 정리 (perceptron convergence theorem)** 으로 증명됨.
  그러나, 비선형 분리 문제는 자동으로 학습할 수 없음

데이터 주도 학습

- 기계학습은 데이터가 생명
- 데이터에서 답을 찾고 데이터에서 패턴을 발견하고 데이터로 이야기를 만듦
- 기계학습의 중심에는 **데이터** 가 존재
- 그러나, 어떤 문제를 해결하려 할 때, 특히 패턴을 찾아내야 할 때는 사람이 이것저것 생각하고 답을 찾는 것이 일반적임. 사람의 경험과 직관을 단서로 시행착오를 거듭하여 일을 진행함
- 기계학습에서는 사람의 개입을 최소화하고 수집한 데이터로부터 패턴을 찾으려고 시도함

- 신경망과 딥러닝은 기존 기계학습에서 사용하던 방법보다 사람의 개입을 더욱 배제할 수 있는 중요한 특성을 지님



- ‘5’를 인식하는 알고리즘을 밑바닥부터 ‘설계’ 하는 대신, 주어진 데이터를 잘 활용해 이미지에서 **특징(feature)**을 추출하고, 그 특징의 패턴을 기계학습 기술로 학습하는 방법이 있음

- 특징(입력 데이터, 즉 입력 이미지)에서 본질적인 데이터(중요한 데이터)를 정확하게 추출할 수 있도록 설계된 것

- 이미지의 특징은 보통 **벡터**로 기술하고, 컴퓨터 비전 분야는 SIFT, SURF, HOG 등의 특징을 많이 사용함

- 이런 특징을 사용하여 이미지 데이터를 벡터로 변환하고, 변환된 벡터를 가지고 지도 학습 방식의 대표 분류 기법인 SVM, KNN 등으로 학습할 수 있음

- 기계 학습에서는 모아진 데이터로부터 규칙을 찾아내는 역할을 ‘기계’ 가 담당함

다만, 이미지를 벡터로 변환할 때 사용하는 특징은 ‘사람이’ 설계함

즉, **문제에 적합한 특징을 설계하지 않으면** 좋은 결과를 얻을 수 없음

- 예를 들어, 강아지의 얼굴을 구분하려 할때는 숫자를 인식할 때와는 다른 특징을 ‘사람’이 생각해야 함. 특징과 기계학습을 활용한 접근에도 문제에 따라서는 ‘사람’이 적절한 특징을 생각해 내야함

- 신경망은 이미지를 '있는 그대로' 학습함
- 두 번째 접근 방식인 특징과 기계학습 방식은 사람이 설계했지만, 신경망은 이미지에 포함된 중요한 특징까지도 '기계'가 스스로 학습함
- 딥러닝을 `**종단간 기계학습(end-to-end machine learning)**` 이라고 함

  종단간 '처음부터 끝까지' 라는 의미로, 데이터(입력) 에서 목표한 결과(출력)을 사람의 개입 없이 얻음
- 신경망의 이점은 모든 문제를 같은 맥락에서 풀 수 있다는 점

  '5'를 인식하는 문제든, '개'를 인식하는 문제든, '사람의 얼굴' 을 인식하는 문제든, 세부 사항과 관계없이 신경망은 주어진 데이터를 온전히 학습하고, 주어진 문제의 패턴을 발견하려 시도함
- 즉, 신경망은 모든 문제를 주어진 데이터 그대로를 입력 데이터로 활용해 'end-to-end'로 학습

훈련 데이터와 시험데이터

- 기계학습 문제는 데이터를 **훈련 데이터(training data)** 와 **시험 데이터(test data)** 로 나눠 학습과 실험을 수행하는 것이 일반적
  훈련 데이터만 사용하여, 학습하면서 최적의 매개변수를 찾음
  다음 시험 데이터를 사용하여 앞서 훈련한 모델의 실력을 평가함
- 우리가 원하는 것은 범용적으로 사용할 수 있는 모델이므로, 범용 능력을 평가하기 위해 훈련 데이터와 시험 데이터를 분리함

- 범용 능력은 아직 보지 못한 데이터(훈련 데이터에 포함되지 않는 데이터)로도 문제를 올바르게 풀어내는 능력이며, 범용 능력을 획득하는 것이 기계학습의 최종 목표임

  예를 들어, 손글씨 숫자 인식의 최종 결과는 엽서에서 우편 번호를 자동으로 판독하는 시스템에 쓰이며, 손글씨 숫자 인식은 '누군가'가 쓴 글자를 인식하는 능력이 높지 않으면 안됨
  '특정인의 특정 글자'가 아니라 '임의의 사람의 임의의 글자' 를 인식해야 함

- 데이터셋 하나로만 매개변수의 학습과 평가를 수행하면 올바른 평가가 될 수 없음.

  한 데이터셋만 지나치게 최적화된 상태인 **오버피팅(overfitting)** 이 발생할 수 있음

4-2. 손실함수

- 신경망 학습에서는 현재의 상태를 '하나의 지표'로 표현하고, 해당 지표를 가장 좋게 만들어주는 가중치 매개변수의 값을 탐색함
- '하나의 지표'를 기준으로 최적의 매개변수 값을 탐색하는데, 이 때 사용하는 지표가 `손실 함수(loss function)`
- 손실함수는 임의의 함수를 사용할 수도 있지만 일반적으로 **오차 제곱합**과 **교차 엔트로피 오차**를 사용
- 손실 함수는 신경망의 성능의 '나쁨'을 나타내는 지표, 현재 신경망이 훈련 데이터를 얼마나 잘 처리하지 '못' 하는지에 대해 나타냄
  손실 함수에 마이너스를 곱하면 '얼마나 나쁘지 않나', '얼마나 좋은가?' 지표로 변환 가능

`오차 제곱합`

오차제곱합(sum of squares for error, SSE)

- 가장 많이 쓰이는 손실 함수는 **오차 제곱합(sum of squares for error, SSE)**

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

- yk : 신경망의 출력(신경망이 출력한 값), tk : 정답 레이블, k는 데이터의 차원수

```
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' 일때의 값
- 신경망의 출력 y는 소프트맥스 함수의 출력은 확률로 해석할 수 있으므로,
  위에서 이미지가 '0'일 확률은 0.1, '1' 일 확률은 0.05 임
- 정답 레이블인 t는 정답을 가리키는 위치의 원소는 1로, 그 외는 0으로 표기함
  여기서는 숫자 '2'에 해당하는 원소의 값이 1이므로 정답이 '2임
- 한 원소만 1로 하고 그 외는 0으로 나타내는 표기법을 **원-핫 인코딩** 이라고 함

- 오차 제곱합은 각 원소의 출력(추정 값)과 정답 레이블(참 값)의 차 (yk - tk)를 제곱한 후, 그 총합을 구함

[ ]:
```
import numpy as np
```

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

[ ]:
```
# 정답은 2
t= [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

# 예1 : '2'일 확률이 가장 높다고 추정함 (0.6)
y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print(sum_squares_error(np.array(y1), np.array(t)))


#예2 : '7'일 확률이 가장 높다고 추정함 (0.6)
y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print(sum_squares_error(np.array(y2), np.array(t)))

```

```
0.09750000000000003
0.5975
```

- 첫 번째의 예는 정답이 '2'고, 신경망의 출력도 '2'에서 가장 높은 경우
- 두 번째 예는 정답이 '2'지만, 신경망 출력은 '7'에서 가장 높은 경우

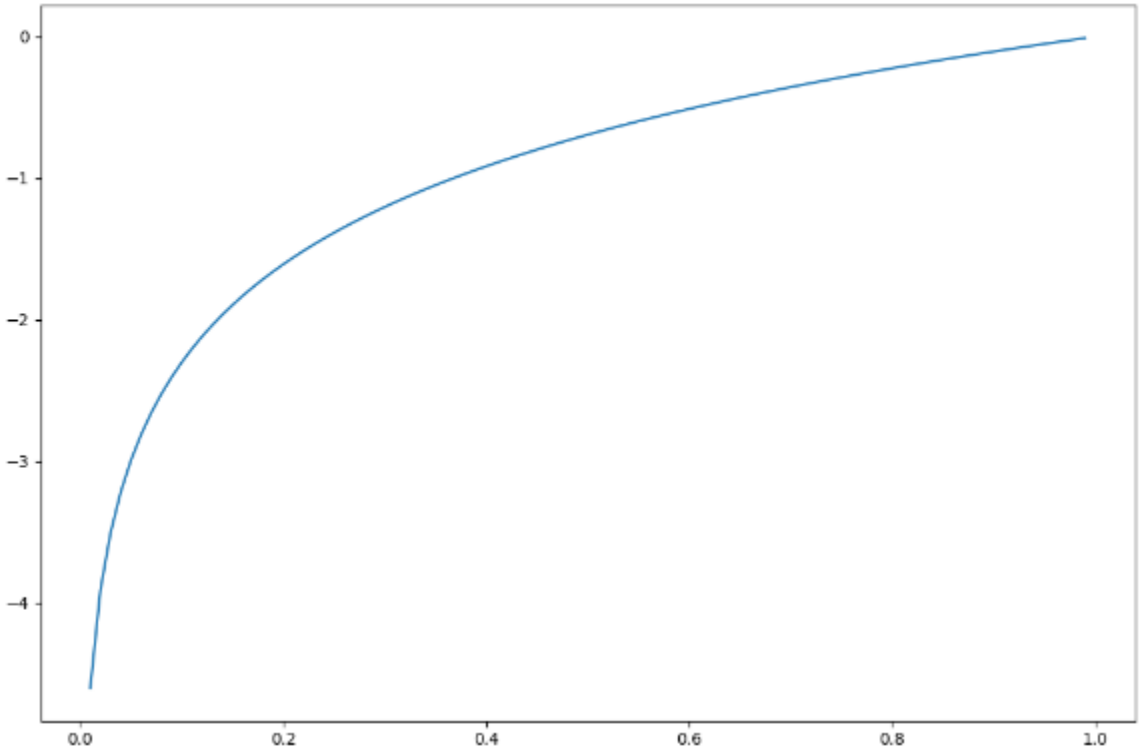
- 첫 번째 예의 손실 함수 쪽 출력이 작으며, 정답 레이블과의 오차도 작은 것을 알 수 있음. 즉, 오차제곱합 기준으로는 첫 번째 추정 결과가(오차가 더 작아서) 답에 가까울 것으로 판단

교차 엔트로피 오차

Cross entropy error, CEE

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

- log는 밑이 e인 자연로그(loge), yk는 신경망의 출력, tk는 정답 레이블
- tk는 정답에 해당하는 인덱스의 원소만 1이고 나머지는 0(원-핫 인코딩)
- 실질적으로 정답일 때의 추정(tk가 1 일때의 yk)의 자연로그 계산
- 예를 들어, 정답 레이블이 '2' 이고, 신경망 출력이 0.6 이라면,
교차 엔트로피 오차는 -log0.6 = 0.51 , 같은 조건에서 신경망 출력이 0.1 이라면 -log0.1 = 2.30임
- **교차 엔트로피 오차는 정답일 때의 출력이 전체 값을 정하게 됨**



[자연로그 y= logx 의 그래프]

- x가 1일 때 y는 0이 되고, x가 0에 가까워질수록 y 값은 점점 작아짐
- 정답에 해당하는 출력이 커질수록 0에 다가가다가, 그 출력이 1일 때 0이 됨
- 반대로 정답일 때의 출력이 작아질수록 오차는 커짐

교차 엔트로피

```
def cross_entropy_error(y,t):
    delta = 1e-7
    return -np.sum(t * np.log(y+delta))
```

```
t = [0,0,1,0,0,0,0,0,0,0]
y1 = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
print(cross_entropy_error(np.array(y1), np.array(t)))

y2 = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
print(cross_entropy_error(np.array(y2), np.array(t)))
```

```
0.510825457099338
2.302584092994546
```

⇒ 아주 작은 값인 delta를 더함(np.log() 함수에 0을 입력하면, 마이너스 무한대 -inf가 나옴)

아주 작은 값을 더해서 절대 0이 되지 않도록, 마이너스 무한대가 발생하지 않도록 함

정답일 때의 출력이 0.6인 경우로, 이때의 교차 엔트로피 오차는 약 0.51

다음은 정답일 때의 출력이 (더 낮은) 0.1인 경우로, 이때의 교차 엔트로피 오차는 2.3

결과(오차 값)가 더 작은 첫 번째 추정이 정답일 가능성이 높다고 판단한 것

미니배치 학습

- 기계학습 문제는 훈련 데이터를 사용하여 학습함
- 훈련 데이터에 대한 손실 함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아냄
- 그렇게 하기 위해서는 모든 훈련 데이터를 대상으로 손실 함수 값을 구해야 함
  즉, 훈련 데이터가 100개 있으면, 그로부터 계산한 100개의 손실 함수 값들의 합을 지표로 삼음

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

- 데이터가 N개라면 t(nk)는 n번째 데이터의 k번째 값을 의미함

  y(nk)는 신경망의 출력, t(nk)는 정답 레이블

- 데이터 하나에 대한 손실 함수를 단순히 N개의 데이터로 확장한 것

- 마지막에 N으로 나눔으로써 정규화하고, '평균 손실 함수'를 구함

- MNIST 데이터셋은 훈련 데이터가 60,000개로, 모든 데이터를 대상으로 손실 함수의 합을 구하려면 시간이 걸림.
- 많은 데이터를 대상으로 일일이 손실 함수를 계산하는 것은 현실적이지 않아, 데이터 일부를 추려 전체의 **근사치**로 이용함
- 신경망 학습에서도 훈련 데이터로부터 일부만 골라 학습을 수행하며, 이 일부를 `미니 배치 (mini-batch)` 라고 함

  예) 60,000장의 훈련 데이터에서 100장을 무작위로 뽑아 그 100장만을 사용하여 학습

```python
import sys, os
import numpy as numpy
import tensorflow as tf

(x_train, t_train), (x_test, t_test) = tf.keras.datasets.mnist.load_data()

print(x_train.shape)
print(t_train.shape)
print(x_test.shape)
print(t_test.shape)
```

```
(60000, 28, 28)
(60000,)
(10000, 28, 28)
(10000,)
```

```python
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)


print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

```python
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]

print(train_size)
print(batch_mask)
print(x_batch)
print(t_batch)
```

```
60000
[29046 30927 57678 47344 20531 54719 23473 35857 14758 33412]
[[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 0 ... 0 0 0]]
[4 4 8 0 0 1 1 0 1 3]
```

- 훈련 데이터는 60,000개이고 입력 데이터는 784열(28x28)
- 정답 레이블은 10줄로된 데이터
- np.random.choice()로 지정한 범위 수 중에서 무작위로 원하는 개수만 꺼냄
  예) np.random.choice(660000,10) 은 0이상 60000 미만의 수 중에서 무작위로 10개를 골라냄
- 무작위로 뽑은 인덱스로 미니배치를 뽑아내고, 손실함수도 이 미니배치로 계산함

- 미니배치의 손실 함수도 일부 표본 데이터로 전체를 비슷하게 계측함
- 즉, 전체 훈련 데이터의 대표로서 무작위로 선택한 **미니배치**를 사용

(배치용) 교차 엔트로피 오차 구현

(배치용) 교차 엔트로피 오차 구하기

```python
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
```

- y : 신경망 출력, t : 정답 레이블
- y가 1차원이라면, 데이터 하나당 교차 엔트로피 오차를 구할 경우 reshape로 데이터 형상 바꿔줌
- 배치 크기로 나눠 정규화후 이미지 1장당 평균의 교차 엔트로피 오차를 계산

```python
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)
```

- 원-핫 인코딩일 때 t가 0인 원소는 교차 엔트로피 오차도 0이므로 계산은 무시해도 좋다는 것이 핵심
- 정답에 해당하는 신경망의 출력만으로 교차 엔트로피 오차를 계산
- 원-핫 인코딩 시 t*np.log(y) 였던 부분을 레이블 표현일 때는 np.log(y[np.arange(batch_size),t])로 구현
- np.log(y[np.arange(batch_size),t])
- np.arange(batch_size)는 0부터 batch_size -1까지의 배열 생성,
  batch_size가 5면, np.arange(batch_size)는 [0,1,2,3,4] 넘파이 배열 생성
  t에는 [2,7,0,9,4]와 같이 저장되어 있으므로 y[np.range(batch_size), t]는 각 데이터의 정답 레이블에 해당하는 신경망의 출력을 추출
  : 이 예에서는 y[np.ragne(batch_size), t]는 [y[0,2], y[1,7], y[2,0], y[3,9], y[4,4]])인 넘파이 배열 생성

손실 함수의 설정

- 궁극적인 목적은 '높은 정확도'를 끌어내는 매개변수 값을 차즌 것
- '정확도' 라는 지표를 놔두고 '손실 함수의 값' 이라는 우회적인 방법을 택하는 것은

  신경망 학습에서의 '미분'의 역할에 주목
- 신경망 학습에서는 최적의 매개변수(가중치와 편향)을 탐색할 때 손실함수의 값을 가능한 작게 하는 매개변수 값을 찾음
- 이 때 매개변수의 미분(정확히는 기울기)을 게산하고, 그 미분 값을 단서로 매개변수의 값을 서서히 갱신하는 과정을 반복
- 가상의 신경망이 있고, 그 신경망의 어느 한 가중치 매개변수에 주목한다고 했을 때, 가중치 매개변수의 손실 함수의 미분은 '가중치 매개변수의 값을 아주 조금 변화 시켰을

때, 손실 함수가 어떻게 변하나'라는 의미

- 만약, 이 미분 값이 음수면 그 가중치 매개변수를 양의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있고, 미분 값이 양수면 가중치 매개변수를 음의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있음
  미분의 값이 0이면 가중치 매개변수를 어느 쪽으로 움직여도 손실 함수의 값은 줄어들지 않아, 가중치 매개변수의 갱신을 멈춤
- **정확도를 지표로 삼아서 안되는 이유는, 미분 값이 대부분의 장소에서 0이 되어 매개변수를 갱신할 수 없기 때문임**

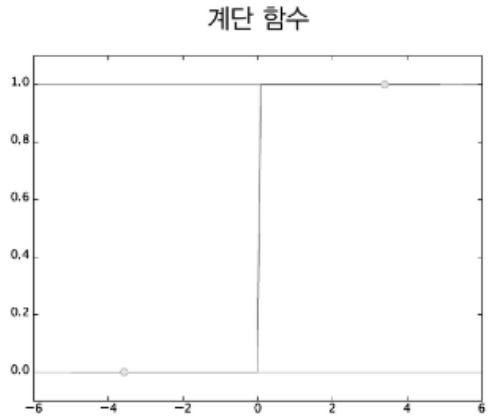
  신경망을 학습할 때 정확도를 지표로 삼아서는 안 된다.
  정확도를 지표로 하면 매개변수의 미분이 대부분의 장소에서 0이 되기 때문임
- 정확도를 지표로 삼으면 매개변수의 미분이 대부분의 장소에서 0이 되는 이유?
  - 한 신경망이 100장의 훈련 데이터 중 32장을 올바로 인식한다고 할 때, 정확도는 32% 이고 정확도가 지표였다면 가중치 매개변수의 값을 조금 바꾼다고 해도 정확도는 그대로 32%임
    즉, 매개변수를 약간만 조정해서는 정확도가 개선되지 않고 일정하게 유지되고, 정확도가 개선된다고 하더라도 값이 32.0123%와 같은 연속적인 변화보다는 33%나 34% 처럼 불연속적인 띄엄띄엄한 값으로 바뀌어버림
  - 손실 함수를 지표로 삼는다면? 현재의 손실 함수의 값이 0.92543 같은 수치로 나타나고, 매개변수의 값이 조금 변하면 그에 반응하여 손실 함수의 값도 0.93432 처럼 연속적으로 변화함
  - 정확도는 매개변수의 미소한 변화에는 거의 반응을 보이지 않고, 그 값이 불연속적으로 갑자기 변화함

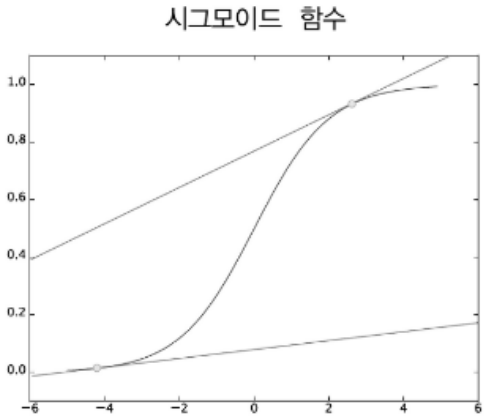
  ⇒ '계단 함수'를 활성화 함수로 사용하지 않는 이유와 같음

  계단 함수의 미분은 대부분의 장소(0이외의 곳)에서 0임

  계단 함수를 이용하면 손실 함수를 지표로 삼는 게 아무 의미가 없어짐

  매개변수의 작은 변화가 주는 파장을 계단 함수가 말살하여 손실 함수의 값에는 아무런 변화가 나타나지 않기 때문임





[계단 함수와 시그모이드 함수 : 계단 함수는 대부분의 장소에서 기울기가 0이지만, 시그모이드 함수의 기울기(접선)은 0이 아님]

- 계단 함수는 한순감나 변화를 일으키지만, 시그모이드 함수의 미분(접선)은 출력(세로축의 값)이 연속적으로 변화고 곡선의 기울기도 연속적으로 변함
- 시그모이드 함수의 미분은 어느 장소라도 0이 되지는 않음

  ⇒ 신경망 학습에서 중요한 성질로, 기울기가 0이 되지 않는 덕분에 신경망이 올바르게 학습할 수 있음

4-3. 수치 미분

경사법에서는 기울기(경사) 값을 기준으로 나아갈 방향을 정함

미분

예) 마라톤 선수가 처음부터 10분에서 2kg씩 달렸다고 할때, 이때의 속도는 간단히 2/10 = 0.2 [km/분] 이라고 계산할 수 있음. 즉, 1분에 0.2km 만큼의 속도(변화)로 뛴 것

- 이 마라톤 예에서 '달린 거리'가 '시간'에 대해서 얼마나 변화했는가를 계산함

  10분에 2km를 뛰었다는 것은. 정확하게는 10분 동안의 '평균 속도'를 구한 것
- **미분**은 **특정 순간의 변화량**을 뜻함

  10분이라는 시간을 가능한 줄여 (직적 1분에 달린 거리, 직전 1초에 달린 거리, 직전 0.1초에 달린 거리.. 식으로 간격을 줄여) 한 순간의 변화량(어느 순간의 속도)를 얻는 것
- 미분은 `한순간의 변화량`을 표시

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

좌변 : f(x)의 x에 대한 미분. x에 대한 f(x)의 변화량을 나타내는 기호

우변 : x의 '작은 변화'가 함수 f(x)를 얼마나 변화시키느냐를 의미함

이 때, 시간의 작은 변화인 시간을 뜻하는 h를 한없이 0에 가깝게 한다는 의미로 lim(h→0)로 나타냄

미분 구현

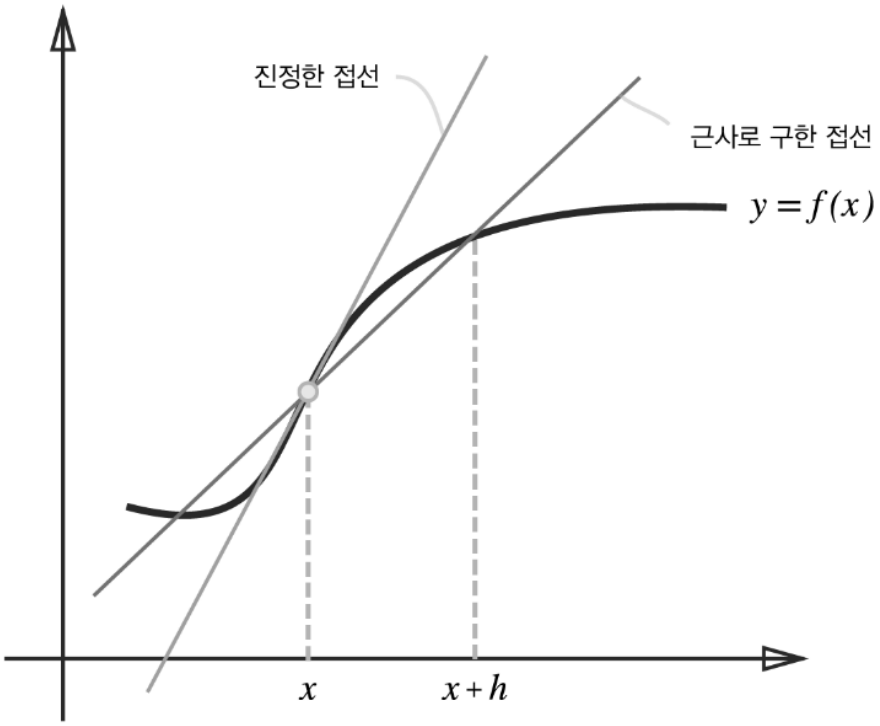
```
# 나쁜 구현

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

- 수치 미분(numerical differenctiation)은 '함수 f'와 '함수 f에 넘길 인수 x' 두 인수를 받음
- h에 가급적 적은 값을 대입하고 싶어서 가능하다면 h를 0을 무한히 가깝게 하고 싶어서 1e-50 이라는 작은 값 이용
 : 해당 방식은 반올림 오차(rounding error) 문제를 일으킴
 : 반올림 오차는 작은 값(가령 소수점 8자리 이하)이 생략되어 최종 계산 결과에 오차가 생김

```
np.float32(1e-50)
```

```
0.0
```

<개선 1>
- 1e-50을 float32형 (32비트 수동소수점)으로 나타내면 0.00이 되어 올바로 표현할 수 없음
- 너무 작은 값을 이용하면 컴퓨터로 계산하는 데 문제가 됨
- 미세한 값 h를 10-4 정도로 사용하면 좋은 결과를 얻음



<개선 2>
- 앞의 구현에는 x+h와 x 사이의 함수 f의 차분을 계산하고 있음
- 진정한 미분은 x 위치의 함수의 기울기(접선)에 해당하지만, 위에서 구현한 미분은 (x+h)와 x 사이의 기울기에 해당함

- 위와 같은 수치 미분은 오차가 포함되어, 오차를 줄이기 위해 (x+h)와 (x-h) 일때의 함수 f의 차분을 계산하는 방법을 사용함
- x를 중심으로 그 전후의 차분을 계산하는 `중심 차분` 혹은 `중앙 차분`

  (x+h)와 x의 차분은 `전방 차분`

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

- 아주 작은 차분으로 미분하는 것을 `수치 미분` 이라고 함
수식을 전개해 미분하는 것은 '해석적(analytic)' 이라는 말을 이용해 '해석적 해', '해석적으로 미분하다' 로 표현
- 예를 들어 y=x^2의 미분은 dy/dx = 2x로 풀어낼 수 있음
  x=2일때 y의 미분은 4가 됨
 => 해석적 미분은 오차를 포함하지 않는 '진정한 미분' 값을 구함

- '해석적 미분'은 수학 시간에 배운 미분,
  '수치 미분'은 이를 '근사치'로 계산하는 방법

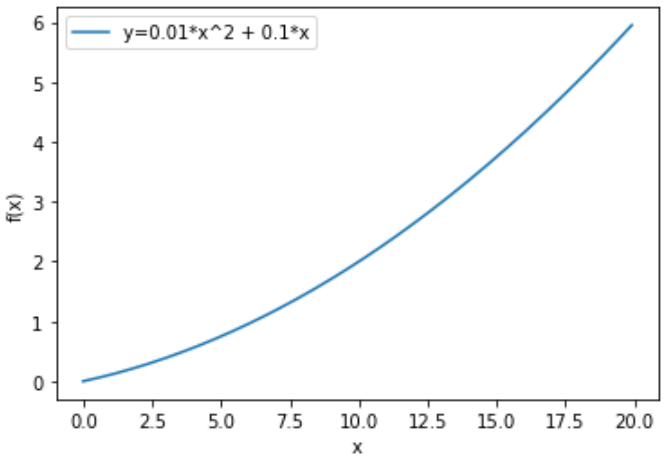
수치 미분의 예

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

수치 미분 구현

```
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 0.1 간격의 배열 x를 만듦(20은 미포함)
y = function_1(x)

plt.xlabel('x')
plt.ylabel('f(x)')
plt.plot(x,y, label = 'y=0.01*x^2 + 0.1*x')
plt.legend()
plt.show()
```




```
# x가 5일 때와 10일 때의 함수의 미분 계산

print(numerical_diff(function_1,5))
print(numerical_diff(function_1,10))
```

```
0.1999999999990898
0.2999999999986347
```

- 이렇게 계산한 미분 값이 x에 대한 f(x)의 변화량인 **함수의 기울기**에 해당
- f(x) = 0.01*x^2 + 0.1*x 의 해석적 해는 df(x)/dx = 0.02*x + 0.1임

  x가 5와 10일 때 '진정한 미분'은 차례로 0.2, 0.3 임

⇒ 위의 수치 미분과 결과를 비교하면 오차가 매우 작음

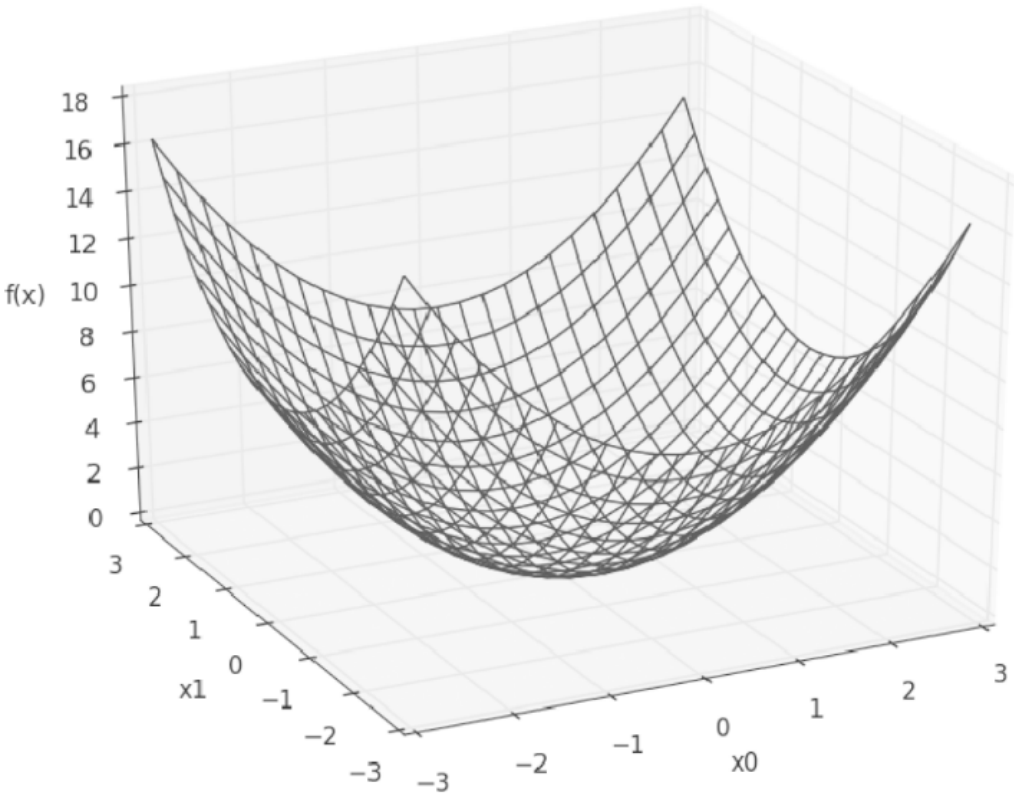
편미분

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

편미분

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

- 인수 x는 넘파이 배열이라고 가정
- 넘파이 배열의 각 원소를 제곱하고 합을 구하는 구현 (np.sum(x**2))



- 위의 식을 미분할 때 주의 점은 변수가 2개인 것임
- '어느 변수에 대한 미분인지' 에 따라 x0, x1 중 어떤 변수인지 구별해야 함

- 변수가 여럿인 함수에 대한 미분을 **편미분**
-

`x0 = 3, x1 = 4 일 때, x0에 대한 편미분 ∂f/∂x0`

```
def function_tmp1(x0):
    return x0*x0 + 4.0**2.0

print(numerical_diff(function_tmp1, 3.0))
```

```
6.00000000000378
```

`x0 = 3, x1 = 4 일 때, x1에 대한 편미분 ∂f/∂x1`

```
def function_tmp2(x1):
    return 3.0**2.0 + x1*x1

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

```
7.999999999999119
```

- 편미분은 변수가 하나인 미분과 마찬가지로 특정 장소의 기울기를 구함
- 단, 여러 변수 중 목표 변수 하나에 초점을 맞추고 다른 변수의 값을 고정함
- 목표 변수를 제외한 나머지를 특정 값에 고정하기 위해서 새로운 함수를 정의
- 그리고 새로 정의한 함수에 대해 사용한 수치 미분 함수를 적용하여 편미분 구함

4-4. 기울기

- x0과 x1의 편미분을 동시에 계산하고 싶다면?
  x0=3, x1=4 일 때, (x0, x1) 양쪽의 편미분을 묶어서 (∂f/∂x0, ∂f/∂x1) 처럼
  모든 변수의 편미분을 벡터로 정리한 것을 `기울기(gradient)` 라고 함

기울기 구현

```python
def numerical_gradient(f, x):
    h = 1e-4  # 0.0001
    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
```

- numerical_gradient(f,x) 함수의 인수인 f는 함수이고, x는 넘파이 배열
넘파이 배열 x의 각 원소에 대해서 수치 미분을 구함
- 세 점 (3,4), (0,2), (3,0) 에서의 기울기 구함

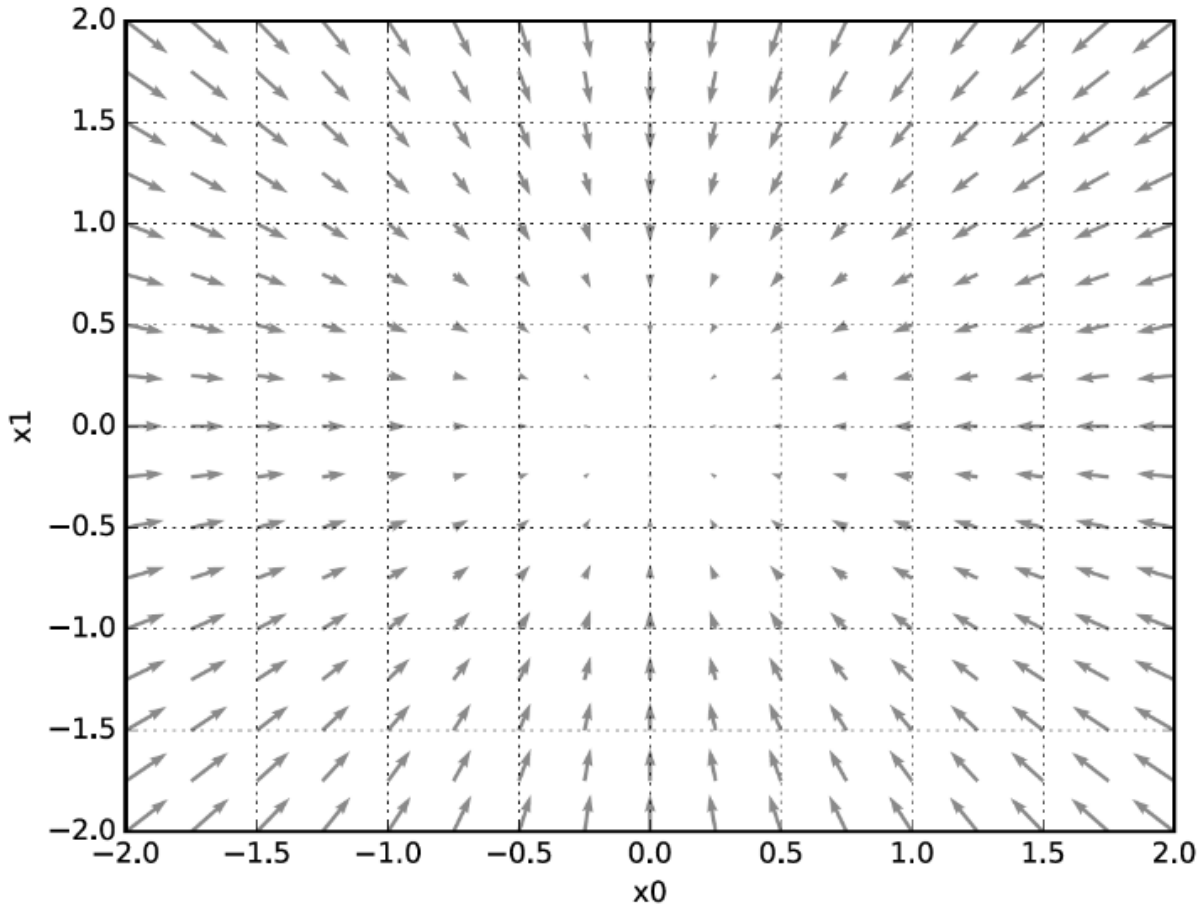
```python
print(numerical_gradient(function_2, np.array([3.0, 4.0])))
print(numerical_gradient(function_2, np.array([0.0, 2.0])))
print(numerical_gradient(function_2, np.array([3.0, 0.0])))
```

```
[6. 8.]
[0. 4.]
[6. 0.]
```

- (x0, x1)의 각 점에서의 기울기를 계산
- 점 (3,4)의 기울기는 (6,8) / 점 (0,2)의 기울기는 (0,4) / 점 (3,0)의 기울기는 (6,0)

- 기울기 그림은 방향을 가진 벡터(화살표)로 그려짐
- 기울기는 함수의 가장 낮은 장소(최솟값)을 가르킴
  화살표들은 한 점을 향하고 있고, '가장 낮은 곳'에서 멀어질수록 화살표의 크기가 커짐
- 기울기는 가장 낮은 장소를 가르키지만, 실제는 반드시 그렇지 않음

  기울기는 각 지점에서 낮아지는 방향을 가리키며 **기울기가 가르키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향** 임



경사법(경사 하강법)

- 기계학습 문제 대부분은 학습 단계에서 최적의 매개변수를 찾아냄
- 신경망 역시 최적의 매개변수(가중치와 편향)를 학습 시에 찾아내야 함
- '최적' 은 손실 함수가 최솟값이 될 때의 매개변수 값
- 일반적인 문제의 손실 함수는 매우 복잡해서, 매개변수 공간이 광대하여 어디가 최솟값인지 짐잘 할 수 없어, 기울기를 잘 이용해 함수의 최솟값(또는 가능한 한 작은 값)을 찾으려는 것이 **경사법**
- 각 지점에서 **함수의 값을 낮추는 방안을 제시하는 지표가 기울기임**
- 그러나, 기울기가 가르키는 곳에 정말 함수의 최솟값이 있는지, 그 쪽이 정말로 나아갈 방향인지를 보장할 수 없음 : 실제로 복잡한 함수에서는 기울기가 가리키는 방향에 최솟값이 없는 경우가 대부분

💡 함수가 극솟값, 최솟값, 또 안장점(saddle point)이 되는 장소에서는 기울기가 0임
극솟값은 국소적인 최소값, 즉 한정된 범위에서의 최솟값인 점
안장점은 어느 방향에서 보면 극댓값이고 다른 방향에서 보면 극솟값이 되는 점

경사법은 기울기가 0인 장소를 찾지만 그것이 반드시 최솟값이라고 할 수는 없음 (극솟값이나 안장점일 가능성이 있음)
복잡하고 찌그러진 모양의 함수라면 평평한 곳으로 파고들면서 고원(plateau, 플래토) 라는 학습이 진행되지 않는 정체기에 빠질 수 있음

- 기울어진 방향이 꼭 최솟값을 가리키는 것은 아니지만, 그 방향으로 가야 함수의 값을 줄일 수 있음
- 최솟값이 되는 장소를 찾는 문제(가능한 작은 값이 되는 장소를 찾는 문제)에서는 기울기 정보를 단서로 나아갈 방향을 정해야 함

경사법

- 현 위치에서 기울어진 방향으로 일정 거리만큼 이동함
  이동한 곳에서도 기울기를 구하고, 기울어진 방향으로 나아가는 것을 반복함

  이렇게 함수의 값을 점차 줄이는 것이 `경사법(gradient method)`
- 경사법은 기계학습을 최적화하는데 흔히 쓰이는 방법,
  신경망 학습에는 경사법을 많이 사용함

> 경사법은 최솟값을 찾느냐, 최댓값을 찾느냐에 따라
> **경사 하강법(gradient descent method)**, **경사 상승법(gradient ascent method)**
>
> 손실 함수의 부호를 반전시키면 최솟값을 찾는 문제와 최댓값을 찾는 문제는 같아서
> 하강이냐/상승이냐는 본질적으로 중요하지 않음

> 일반적으로 신경망(딥러닝) 분야에서의 경사법은 `경사하강법` 으로 등장

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

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

η 기호(에타)는 갱신하는 양을 나타냄

⇒ 신경망 학습에서의 `학습률(learning rate)`

- 학습률 : 한 번의 학습으로 얼마만큼 학습할 지, 매개변수 값을 얼마나 갱신할지 정함
- 위의 식은 1회에 해당하는 갱신이고, 이 단계를 반복하면서 서서히 함수의 값을 줄임
- 학습률의 값은 0.01, 0.001 등 미리 특정값으로 정해두어야 함

  → 이 값이 너무 크거나 작으면 '좋은 장소'를 찾아갈 수 없음

  → 신경망 학습에서는 이 학습률 값을 변경하면서 올바르게 학습하고 있는지를 확인하면서 진행함

`경사하강법 구현`

```python
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
```

- 인수 f는 최적화 하려는 함수, init_x : 초기값, lr : learning rate를 의미하는 학습률
step_num : 경사법에 따른 반복 횟수
- 함수의 기울기는 numerical_gradient(f,x)로 구하고, 기울기에 학습률을 곱한 값으로 갱신하는 처리를 step_num번 반복

경사법으로 f(x0, x1)=x0^2 +x1^2 의 최솟값 구하기

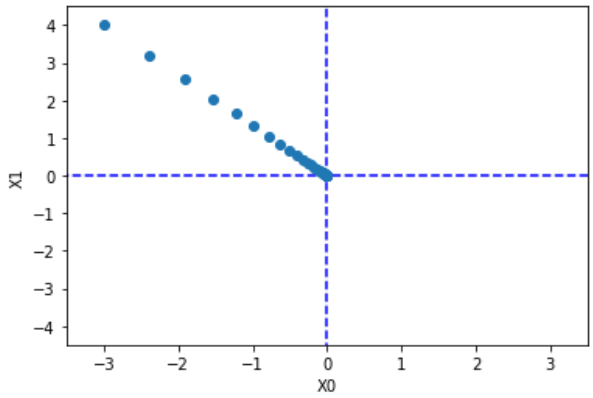
```
def function_2(x):
    return x[0]**2 + x[1]**2

init_x = np.array([-3.0, 4.0])
print(gradient_descent(function_2, init_x=init_x, lr=0.1, step_num=100))
```

```
[-6.11110793e-10  8.14814391e-10]
```

```
x, x_history = gradient_descent_graph(function_2, init_x= np.array([-3.0, 4.0]), lr=0.1, step_num=100)

print(x_history)
```

```
[[-3.00000000e+00  4.00000000e+00]
 [-2.40000000e+00  3.20000000e+00]
 [-1.92000000e+00  2.56000000e+00]
 [-1.53600000e+00  2.04800000e+00]
 [-1.22880000e+00  1.63840000e+00]
 [-9.83040000e-01  1.31072000e+00]
 [-7.86432000e-01  1.04857600e+00]
 [-6.29145600e-01  8.38860800e-01]
 [-5.03316480e-01  6.71088640e-01]
 [-4.02653184e-01  5.36870912e-01]
 [-3.22122547e-01  4.29496730e-01]
 [-2.57698038e-01  3.43597384e-01]
 [-2.06158430e-01  2.74877907e-01]
 [-1.64926744e-01  2.19902326e-01]
 [-1.31941395e-01  1.75921860e-01]
 [-1.05553116e-01  1.40737488e-01]
 [-8.44424930e-02  1.12589991e-01]
 [-6.75539944e-02  9.00719925e-02]
 [-5.40431955e-02  7.20575940e-02]
```

```
x, x_history = gradient_descent_graph(function_2, init_x= np.array([-3.0, 4.0]), lr=0.1, step_num=100)

plt.plot( [-5, 5], [0,0], '--b')
plt.plot( [0,0], [-5, 5], '--b')
plt.plot(x_history[:,0], x_history[:,1], 'o')

plt.xlim(-3.5, 3.5)
plt.ylim(-4.5, 4.5)
plt.xlabel("X0")
plt.ylabel("X1")
plt.show()
```

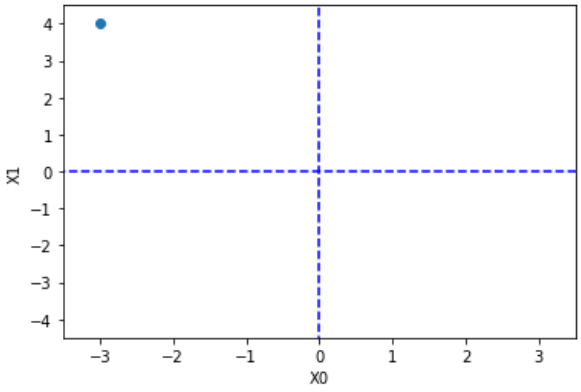


학습률에 따른 비교

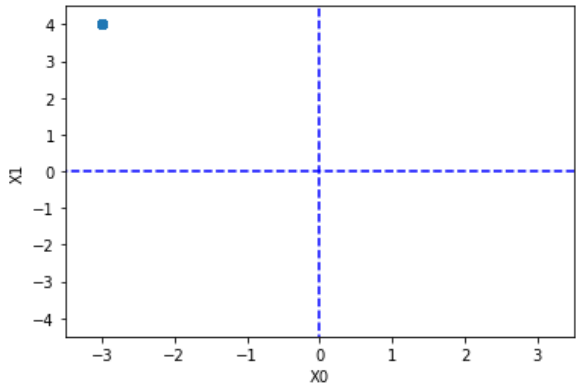
```
# 학습률이 너무 큰 경우 lr = 1.0
init_x = np.array([-3.0, 4.0])
print(gradient_descent(function_2, init_x = init_x, lr=10.0, step_num=100))

# 학습률이 너무 작은 경우 lr = 1e-10
init_x = np.array([-3.0, 4.0])
print(gradient_descent(function_2, init_x = init_x, lr=1e-10, step_num=100))
```

```
[-2.58983747e+13 -1.29524862e+12]
[-2.99999994  3.99999992]
```

```
# 학습률이 너무 큰 경우

x, x_history = gradient_descent_graph(function_2, init_x= np.array([-3.0, 4.0]), lr=10.0, step_num=100)

plt.plot( [-5, 5], [0,0], '--b')
plt.plot( [0,0], [-5, 5], '--b')
plt.plot(x_history[:,0], x_history[:,1], 'o')

plt.xlim(-3.5, 3.5)
plt.ylim(-4.5, 4.5)
plt.xlabel("X0")
plt.ylabel("X1")
plt.show()
```




```
# 학습률이 너무 작은 경우

x, x_history = gradient_descent_graph(function_2, init_x= np.array([-3.0, 4.0]), lr=1e-10, step_num=100)

plt.plot( [-5, 5], [0,0], '--b')
plt.plot( [0,0], [-5, 5], '--b')
plt.plot(x_history[:,0], x_history[:,1], 'o')

plt.xlim(-3.5, 3.5)
plt.ylim(-4.5, 4.5)
plt.xlabel("X0")
plt.ylabel("X1")
plt.show()
```



> 학습률 같은 매개변수를 `하이퍼파라미터(hyper parameter, 초매개변수)` 라고 함
> 가중치와 편향 같은 신경망의 매배견수와는 성질이 다른 매개변수
> 신경망의 가중치 매개변수는 훈련 데이터와 학습 알고리즘에 의해서 '자동'으로 획득되는 매개변수인 반면, 학습률 같은 하이퍼파라미터는 사람이 직접 설정해야 하는 매개변수
> 하이퍼파라미터들은 여러 후보 값 중에서 시험을 통해 가장 잘 학습하는 값을 찾는 과정을 거쳐야 함

신경망에서의 기울기

- 신경망 학습에서도 기울기를 구해야 하는데, 기울기는 **가중치 매개변수에 대한 손실 함수의 기울기**

  예를 들어 형상이 **2X3**, 가중치가 **W**, 손실함수가 **L**인 신경망은 경사를 **∂L/∂W** 로 나타냄

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{pmatrix}$$

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{31}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{32}} \end{pmatrix}$$

  - **∂L/∂W** 의 각 원소는 각각의 원소에 관한 편미분

  1행 1번째 원소인 **∂L/∂W(11) 은** w11 을 조금 변경했을 때 손실 함수 L 이 얼마나 변화하느냐를 나타냄

  - **∂L/∂W**의 형상이 W와 같음. W와 **∂L/∂W**의 형상은 모두 2x3임

기울기를 구하는 simpleNet

```
def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T

    x = x - np.max(x) # 오버플로 대책
    return np.exp(x) / np.sum(np.exp(x))

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

    # 훈련 데이터가 원-핫 벡터라면 정답 레이블의 인덱스로 반환
    if t.size == y.size:
        t = t.argmax(axis=1)

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

```python
def _numerical_gradient_1d(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    for idx in range(x.size):
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

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

        x[idx] = tmp_val # 값 복원

    return grad


def numerical_gradient_2d(f, X):
    if X.ndim == 1:
        return _numerical_gradient_1d(f, X)
    else:
        grad = np.zeros_like(X)

        for idx, x in enumerate(X):
            grad[idx] = _numerical_gradient_1d(f, x)

        return grad

def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

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

        x[idx] = tmp_val # 값 복원
        it.iternext()

    return grad
```

simpleNet 클래스

```
class simpleNet():
    def __init__(self):
        self.W = np.random.randn(2,3) # 정규 분포로 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
```

- softmax와 cross_entropy_error, numerical_gradient 메서드도 이용
- simpleNet 클래스는 형상이 2x3 인 가중치 매개변수 하나를 인스턴스 변수로 갖음
- 메서드는 2개이며, 하나는 예측을 수행하는 predict(x), 하나는 손실 함수의 값을 구하는 loss(x, t) ⇒ 인수 x는 입력 데이터, t는 정답 레이블

```
net = simpleNet()
print(net.W)

x = np.array([0.6, 0.9])
p = net.predict(x)
print(p)
print(np.argmax(p))

t = np.array([0,0,1]) # 정답 레이블
print(net.loss(x,t))

def f(W):
    return net.loss(x,t)

dW = numerical_gradient(f, net.W)
print(dW)
```

```
[[ 1.67179961  1.33307505  0.60963116]
 [ 0.00561861 -0.92873926  0.25664102]]
[ 1.00813652 -0.0360203   0.59675561]
0
1.11186228111521 57
[[ 0.29780761  0.10482485 -0.40263246]
 [ 0.44671141  0.15723727 -0.60394869]]
```

- 기울기는 numerical_gradient(f,x)를 써서 구함
- 여기서 f(W) 의 함수의 인수 W는 더미(dummy)로 만든 것
- numerical_gradient(f,x) 내부에서 f(x)를 실행함

- numerical_gradient(f,x)의 인수 f는 함수, x는 함수 f의 인수
- net.W를 인수로 받아 손실 함수를 계산하는 새로운 함수 f를 정의
- dW는 numerical_gradient(f, net.W)의 결과로 형상은 2x3의 2차원 배열
  dW에서 예를 들어 **∂L/∂W**의 **∂L/∂W11** 은 대략 **0.2**
- w1을 h만큼 늘리면 손실 함수의 값은 0.2h 만큼 증가
- **∂L/∂W23은 대략 -0.5이니, ∂L/∂W23을 h만큼 틀리면 손실 함수의 값은 0.5h 만큼 감소**
- 손실 함수를 줄인다는 관점에서 **∂L/∂W23은 양의 방향으로 갱신되고, w11은 음의 방향으로 갱신해야 함**
- 갱신되는 양에는 **∂L/∂w23이 w11 보다 크게 기여함**

4-5. 학습 알고리즘 구현하기

[신경망의 학습 절차]

- 전제 : 신경망에는 적응 가능한 가중치와 편향이 있고,
  이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습' 이라고 함
  '신경망 학습'은 4단계로 수행
  - 1단계 - '미니배치'
    - 훈련 데이터 중 일부를 무작위로 가져옴
    - 이렇게 선별한 데이터를 '미니배치'라고 하며, 이 미니배치의 손실 함수 값을 줄이는 것이 목표
  - 2단계 - '기울기 산출'
    - 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구함
    - 기울기는 손실 함수의 값을 가장 작게 하는 방향을 제시
  - 3단계 - '매개변수 갱신'
    - 가중치 매개변수를 기울기 방향으로 아주 조금 갱신
  - 4단계 - '반복'
    - 1~3 단계의 반복

- 경사 하강법으로 매개변수를 갱신하고, 데이터를 미니배치로 무작위로 선정하기 때문에 **확률적 경사 하강법(stochastic gradient descent, SGD)** 라고 부름
- '확률적으로 무작위로 골라낸 데이터'에 대해 수행하는 경사 하강법
- 딥러닝 프레임워크는 확률적 경사 하강법의 머리글자를 딴 **SGD** 함수로 이 기능 구현 가능

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T

    x = x - np.max(x) # 오버플로 대책
    return np.exp(x) / np.sum(np.exp(x))

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

    # 훈련 데이터가 원-핫 벡터라면 정답 레이블의 인덱스로 반환
    if t.size == y.size:
        t = t.argmax(axis=1)

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

```python
def _numerical_gradient_1d(f,x):
    h = 1e-4
    grad = np.zeors_like(x)

    for idx in range(x.size):
        tmp_val = x[idx]
        x[idx] =  float(tmp_val) + h
        fxh1 = f(x)

        x[idx] = float(tmp_val) - h
        fxh2 = f(x)
        grad[idx] = (fxh1 - fxh2) / 2*h

        x[idx] = tmp_val

    return grad


def numerical_gradient_2d(X):
    if X.dim==1:
        return self._numerical_gradient_1d(f, X)

    else:
        grad = np.zeros_like(x)

        for idx, x in enumerate(X):
            grad[idx] = self._numerical_gradeint_1d(f,x)
    return grad

def numerical_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

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

        x[idx] = tmp_val # 값 복원
        it.iternext()

    return grad
```

[2층 신경망 클래스 구현]

```python
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 : 입력데이터, 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 : 입력데이터, t : 정답레이블

    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
```

[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']은 2번째 층의 가중치, params['b2']은 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 편

	향의 기울기 grads['W2']는 2번째 층의 가중치의 기울기, grads['b2']는 2번째 층의 편향의 기울기

[TwoLayerNet 클래스의 메서드]

메서드	설명
init(self, input_size,hidden_size, ouput_size)	초기화 수행 인수는 순서대로 입력층의 뉴런 수, 은닉층의 뉴런 수, 출력층의 뉴런 수
predict(self,x)	예측(추론) 수행 인수 x는 이미지 데이터
loss(self,x,t)	손실 함수의 값을 구함 인수 x는 이미지 데이터, t는 정답 레이블
accuracy(self,x,t)	정확도 구함
numerical_gradient(self,x,t)	가중치 매개변수의 기울기를 구함
gradient(self,x,t)	가중치 매개변수의 기울기를 구함 numerical_gradient() 성능 개선판

- TwoLayerNet 클래스는 딕셔너리인 params와 grads를 인스턴스 변수로 갖음
- params 변수에 가중치 매개변수가 저장
- 1번째 층의 가중치 매개변수는 params['W1'] 키에 넘파이로 저장
- 1번째 층의 편향은 params['b1'] 키로 접근

```
net = TwoLayerNet(input_size= 784, hidden_size=100, output_size=10)

print(net.params['W1'].shape)
print(net.params['b1'].shape)
print(net.params['W2'].shape)
print(net.params['b2'].shape)
```

```
(784, 100)
(100,)
(100, 10)
(10,)
```

- params 변수에 신경망에 필요한 매개변수가 모두 저장됨
- params 변수에 저장된 가중치 매개변수가 예측 처리(순방향 처리)에서 사용됨
- grads 변수에는 params 변수에 대응하는 각 매개변수의 기울기가 저장됨

- numerical_gradient() 메서드를 사용해 기울기를 계산하여 grads 변수에 기울기 정보 저장

```
x = np.random.rand(100,784) # 더미 입력 데이터 100장 분량
t = np.random.rand(100,10) # 더미 정답 데이터 100장 분량

grads = net.numerical_gradient(x,t)

print(grads['W1'].shape)
print(grads['b1'].shape)
print(grads['W2'].shape)
print(grads['b2'].shape)

```

```
(784, 100)
(100,)
(100, 10)
(10,)
```

- **init(self, input_size, hidden_size, ouput_size) 메서드로 클래스를 초기화 함**

- 초기화 메서드는 TwoLayerNet을 생성할 때 불리는 메서드,
인수는 순서대로 입력층의 뉴런 수, 은닉층의 뉴런 수, 출력층의 뉴런 수

예를 들어, 손글씨 숫자 인식에서 크기가 28x28 인 입력 이미지가 총 784개고, 출력은 10개가 됨
따라서 input_size=784, ouput_size=10 으로 젖아하고, 은닉츠의 개수인 hidden_size는 적당한 값 설정

- 초기화 메서드에서는 가중치 매개변수도 초기화함

- 가중치 매개변수의 초깃값을 무엇으로 설정하냐가 신경망 학습의 성공을 좌우하기도 함

- 지금은 정규분포를 따르는 난수로 편향을 0으로 초기화 중

- loss(slef,x,t)는 손실 함수의 값을 계산하며, predict() 결과와 정답 레이블을 바탕으로 교차 엔트로피 오차를 구함

- numerical_Gradient(self, x,y) 메서드는 각 매개변수의 기울기를 계산
수치 미분 방식으로 각 매개변수의 손실 함수에 대한 기울기를 계산

- gradient(self, x, t) 오차역전파법을 사용하여 기울기를 효율적이고 빠르게 계산

numerical_gradeint(self, x, t)는 수치 미분 방식으로 매개변수의 기울기를 계산
이 기울기 계산을 고속으로 수행하는 방법인 오차역전파 법

오차역전파법을 쓰면 수치 미분을 사용할 때와 거의 같은 결과를 훨씬 빠르게 얻을 수 있음

미니배치 학습 구현

- 미니배치 학습은 훈련 데이터 중 일부를 무작위로 꺼내고(미니배치), 그 미니배치에 대해서 경사법으로 매개변수를 갱신
- TwoLayerNet 클래스와 MNIST 데이터셋을 사용하여 학습 수행

시험 데이터로 평가

- 손실 함수의 값은 '훈련 데이터의 미니배치에 대한 손실 함수'의 값
- 훈련 데이터의 손실 함수 값이 작아지는 것은 신경망이 잘 학습하고 있다는 방증
- 그러나, 이 결과로 다른 데이터셋에도 비슷한 실력을 발휘할지 확실하지 않음
- 신경망 학습에서는 훈련 데이터 외의 데이터를 올바르게 인식하는지를 확인해야 함
- '오버피팅'인 훈련 데이터에 포함된 이미지만 제대로 구분하고, 그렇지 않은 이미지를 식별하지 못하는 것을 막아야 함
- 신경망 학습의 원래 목표는 범용적인 능력을 익히는 것
- 범용 능력을 평가하려면 훈련 데이터에 포함되지 않은 데이터를 사용해 평가함
- 학습 도중 정기적으로 훈련 데이터와 시험 데이터를 대상으로 정확도를 기록

→ 1에폭별로 훈련 데이터와 시험 데이터에 대한 정확도 기록

💡 **에폭(epoch)** 은 하나의 단위
1에폭은 학습에서 훈련 데이터를 모두 소진했을 때의 횟수
훈련 데이터 10,000개를 100개의 미니배치로 학습할 경우,
확률적 경사 하강법을 100회 반복하면 모든 훈련 데이터를 '소진' 한 것이 됨
이 경우 100회가 1에폭(epoch)

4-6. 정리

- 신경망이 학습을 수행할 수 있도록 손실 함수라는 지표 도입
- 손실 함수를 기준으로 그 값이 가장 작아지는 가중치 매개변수 값을 찾아내는 것이 신경망 학습의 목표
- 가능한 한 작은 손실 함수의 값을 찾는 수법 → 경사법
- 경사법은 함수의 기울기를 이용하는 방법
- 기계학습에서 사용하는 데이터셋은 훈련 데이터와 시험 데이터로 나눠 ㅅ용함
- 훈련 데이터로 학습한 모델의 범용 능력을 시험 데이터로 평가함
- 신경망 학습은 손실 함수를 지표로, 손실 함수의 값이 작아지는 방향으로 가중치 매개변수 갱신
- 가중치 매개변수를 갱신할 때는 가중치 매개변수의 기울기를 이용하고, 기울어진 방향으로 가중치의 값을 갱신하는 작업을 반복
- 아주 작은 값을 주었을 때의 차분으로 미분하는 것을 수치 미분
- 수치 미분을 이용해 가중치 매개변수의 기울기를 구함
- 수치 미분을 이용한 계산에는 시간이 걸리지만, 구현은 간단함

  복잡한 오차역전파법은 기울기를 고속으로 구할 수 있음