Chapter

3

신경망 학습알고리즘

학습 목표

✓ 학습에 대해 이해한다.
✓ 주요 학습 기술들을 이해한다.
✓ 신경망을 직접 구현한다.

주요 내용

✓ 딥러닝의 기본인 신경망 구현에 필요한 수식 이해
✓ 딥러닝에 사용되는 Loss Function 이해
✓ 신경망 학습 방법 및 알고리즘 구현

3

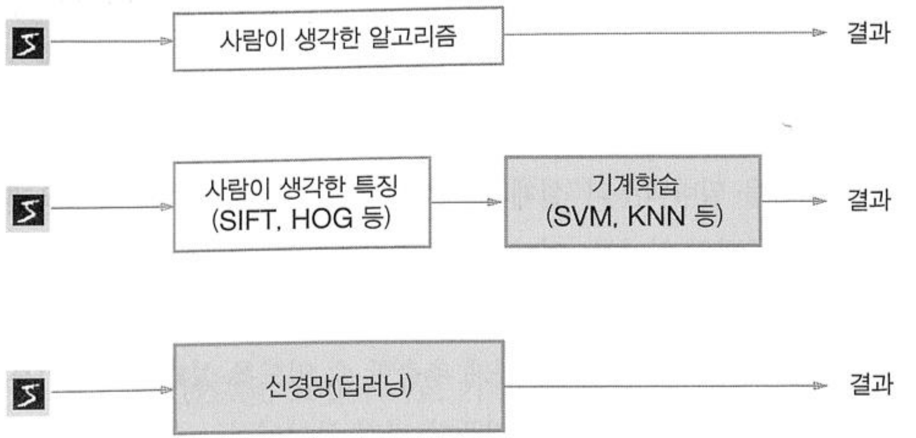
1. 데이터 주도 학습

데이터 주도 학습

❖ 데이터 주도 학습

MNIST 손글씨 데이터

- 학습이란?
  - 훈련데이터로부터 가중치 매개변수의 최적값을 자동으로 획득하는 것
- 데이터 에서 학습한다는 것은?
  - 가중치 매개변수의 값을 데이터를 보고 자동으로 결정하는 것
- 사람 vs 기계학습 vs 신경망. 딥러닝
  - 딥러닝?
    - end-to-end learning(종단간 기계 학습)
    - 데이터 (입력)에서 목표한 결과(출력)을 얻음



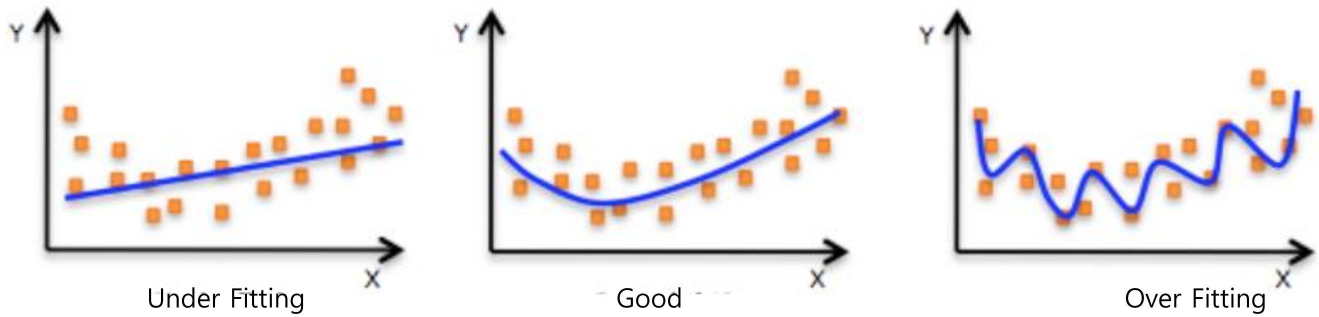
데이터 주도 학습

❖ 데이터 주도 학습

- 훈련 데이터(training data)
    - 훈련 데이터만 사용하여 학습하면서 최적의 매개변수 찾기
- 테스트 데이터(test data)
    - 학습시킨 모델 검증

- 왜 훈련데이터와 테스트 데이터를 나누나요?
    - 범용적으로 사용할 수 있는 모델 구현을 위해 (일반화)

- Overfitting: 하나의 데이터 셋에 지나치게 최적화된 상태



3

2. Loss Function

Loss Function

❖ Loss Function (손실 함수) == Cost Function(비용함수)

- Loss Function (손실 함수): ' 하나의 지표'를 기준으로 최적의 매개변수 값을 탐색
  - 즉, 신경망의 학습 정도를 수치화 하는데 사용되며, 손실 함수의 값을 최소화 하는 방향으로 신경망의 매개변수 값 조정
  - **평균 제곱 오차 (mean squared error, MSE)**
  - **교차 엔트로피 오차 (cross entropy error, CEE)**

Loss Function

❖ Loss Function (손실 함수): 평균 제곱 오차 (MSE)

- 평균 제곱 오차(**mean squared error, MSE**)
- $$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$
- y_k : 신경망의 출력
- t_k : 정답 레이블
- k : 데이터의 차원 수

```
In [1]: import numpy as np

In [4]: y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
        true = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

In [5]: def mean_squared_error(y, t):
            return 0.5 * np.sum((y - t)**2)
```

정답 (t) 와 예측값 (y) 의 오차를 MSE로 계산

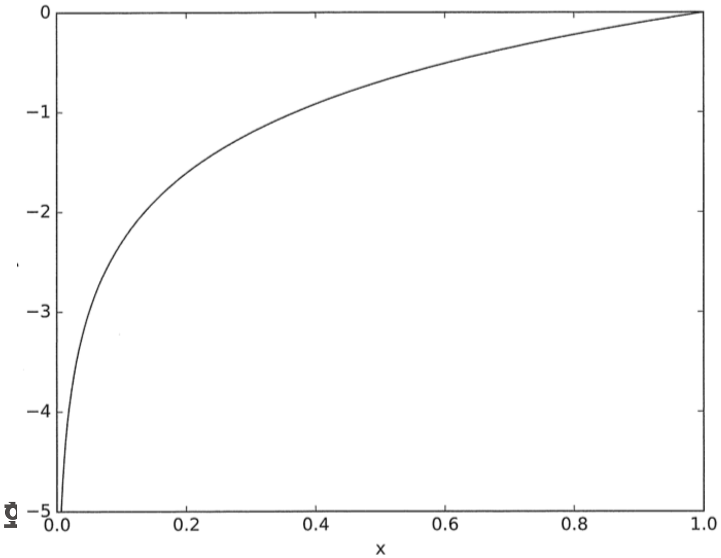
```
In [6]: mean_squared_error(np.array(y), np.array(true))

Out[6]: 0.09750000000000003

In [7]: y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
        mean_squared_error(np.array(y), np.array(true))

Out[7]: 0.5975
```

첫번째 실험의 MSE값이 더 작으므로 더 정답에 가깝다고 판단 가능

Loss Function

❖ Loss Function (손실 함수): 교차 엔트로피 오차 (CEE)

- 교차 엔트로피 오차 **(cross entropy error, CEE)**
- $E = -\sum_{k} t_k \log(y_k)$
- y_k : 신경망의 출력(자연로그)
- t_k : 정답 레이블
    - 정답에 해당하는 인덱스의 원소만 1이고 나머지는 0 (one-hot encoding)
    - 실질적으로 t_k가 1일 때의 y_k의 자연로그를 계산 하는 것



- x가 1일 때, y는 0이 됨
- x가 0에 가까워 질 수록 y값은 점점 작아짐
- 즉, 정답일 수록 0에 가깝고,
- 정답 출력이 작으면 오차가 커짐

$y = logx$ 그래프

Loss Function

❖ Loss Function (손실 함수): 교차 엔트로피 오차 (CEE) Code

- 교차 엔트로피 오차 (CEE)

```
In [8]: def cross_entropy_error(y, t):
            delta = 1e-7                      # log0 방지를 위함
            return -np.sum(t * np.log(y + delta))
```

정답 (t) 와 예측값 (y) 의 오차를 CEE로 계산

```
In [9]: y = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]
        true = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

```
In [10]: cross_entropy_error(np.array(y), np.array(true))

Out[10]: 0.510825457099338
```

```
In [11]: y = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]
         cross_entropy_error(np.array(y), np.array(true))

Out[11]: 2.302584092994546
```

Loss Function

❖ Loss Function (손실 함수): 미니배치 학습

- 배치 학습
  - 훈련 데이터에 대한 손실 함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아내는 것이 목적
  - 훈련 데이터 모두에 대한 손실 함수의 합을 구하는 법? **평균손실함수 이용**
  - $$E = -\frac{1}{N}\sum_{k}\sum_{k} t_{nk} \log y_{nk}$$
    - N: 데이터의 수
    - t_{nk} : 정답 레이블
      - n번째 데이터의 k 차원째의 값
    - y_{nk}: 신경망의 출력

- N(데이터의 총 개수)으로 나누어 정규화
  - '평균 손실 함수' : 데이터의 개수에 상관 없이 통일된 지표 획득 가능

Loss Function

❖ Loss Function (손실 함수): 미니배치 학습

- 미니 배치(mini-batch) 학습
    - 빅데이터 학습
        - MNIST 데이터 셋의 훈련 데이터는 60,000 개
            - 모든 데이터를 대상으로 손실 함수를 구하면? -> 시간이 너무 오래 걸리는 문제
    - 즉, 훈련 데이터로부터 일부만 골라서 학습을 수행하는 방법
        - 이때 일부를 미니배치 라고 함
        - 예: MNIST 데이터의 훈련 데이터인 60,000장 중에서 100장을 무작위로 뽑아서 100장을 사용하여 학습

Loss Function

❖ Loss Function (손실 함수): 미니배치 학습 Code

- 미니 배치(mini-batch) 학습

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
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 :', x_train.shape)
print('t_train.shape :', t_train.shape)
```

```
x_train.shape : (60000, 784)
t_train.shape : (60000, 10)
```

```
train_size = x_train.shape[0]
batch_size = 10
# 0~59999 에서 10개 random하게 추출
batch_mask = np.random.choice(train_size, batch_size)
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]

print('batch_mask :', batch_mask)
print('x_batch.shape :', x_batch.shape)
print('t_batch.shape :', t_batch.shape)
```

무작위로 10개 추출

```
batch_mask : [51942 52877  5460 47153 21975 49011  22218 13330 51696 41146]
x_batch.shape : (10, 784)
t_batch.shape : (10, 10)
```

Loss Function

❖ Loss Function (손실 함수): 미니배치용 CEE 구현하기

- 미니 배치(mini-batch) 학습에 사용하는 교차 엔트로피 오차는 평균손실함수로 계산

one hot encoding 용

```
In [14]: def cross_entropy_error_one_hot(y, t):
             if y.ndim == 1:
                 t = t.reshape(1, t.size) # 정답레이블
                 y = y.reshape(1, y.size) # 신경망의 출력

             batch_size = y.shape[0]
             return -np.sum(t * np.log(y)) / batch_size
```

label 용

```
In [15]: def cross_entropy_error_label(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])) / batch_size
```

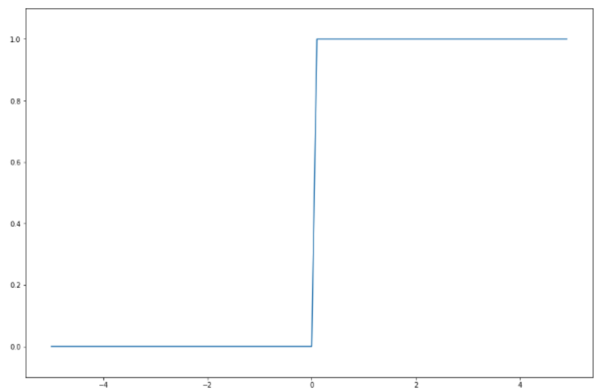
t에 숫자레이블이 주어지면 2차원 배열 형태로 label값을 저장하여 log 처리

Loss Function

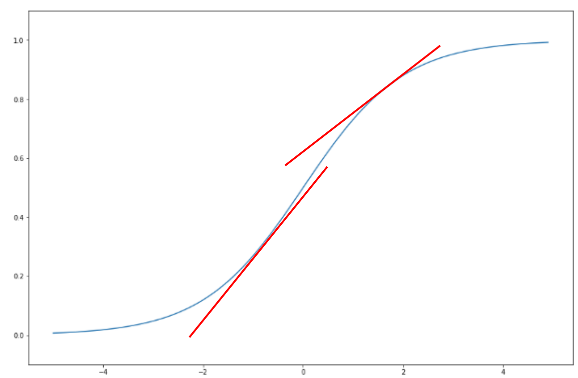
❖ Loss Function (손실 함수): 왜 설정해야 하는가?

- 최적의 매개변수(가중치, 편향) 는 손실 함수의 값을 가장 작게 만드는 매개변수
- 매개변수의 미분(기울기, gradient)을 계산하고, 그 미분 값을 이용하여 매개변수의 값을 서서히 갱신
- 미분 값이 0이 되면 매개변수의 갱신 중단

- 손실함수가 계단함수이면 대부분의 지점에서 기울기가 0이 되어 매개변수의 갱신이 불가능
- 손실 함수가 시그모이드 함수면 기울기가 0이 안됨



Step Function



Sigmoid Function

3

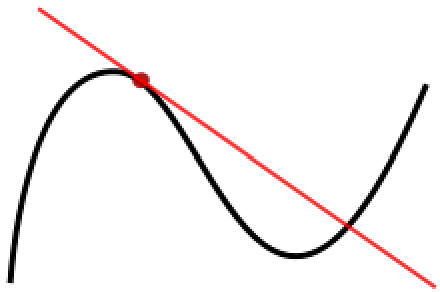
3. Numerical differentiation

Numerical differentiation

❖ 미분

- 경사하강법에서 매개변수의 기울기(미분)의 값을 기준으로 매개변수 조정

- **미분**(derivative) **?**
  - 한 순간의 변화량
  - $$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$
    - $f(x)$의 x에 대한 미분



함수의 그래프와 그 접선 함수의 점에서의 미분은 그 점에서의 접선의 기울기와 같다

https://en.wikipedia.org/wiki/Derivative

In [16]:
```python
def numerical_differential(f, x):
    h = 10e-50
    return (f(x + h) - f(x)) / h
```

Numerical differentiation

❖ 미분

In [16]:
```
def numerical_differential(f, x):
    h = 10e-50
    return (f(x + h) - f(x)) / h
```

- h값이 작으면 반올림 오차 문제 발생 -> $h = 10^{-4}$이 좋은 결과를 낸다고 알려짐

- $(x+h)$와 x 사이의 기울기와 x의 접선의 기울기가 달라 현재 계산 방식에 오차 존재
  - $(x+h)$와 $(x-h)$의 차이를 $2h$로 나눠서 해결

$$h = 0.0001$$
$$\frac{f(x+h) - f(x-h)}{2h}$$

In [20]:
```
def numerical_differential(f, x):
    h = 1e-4
    return (f(x + h) - f(x-h)) / (2*h)
```

Numerical differentiation

❖ 수치 미분

- $y = 0.01x^2 + 0.2x$ 를 미분

```
In [21]: def function_1(x):
             return 0.01*x**2 + 0.1*x
```

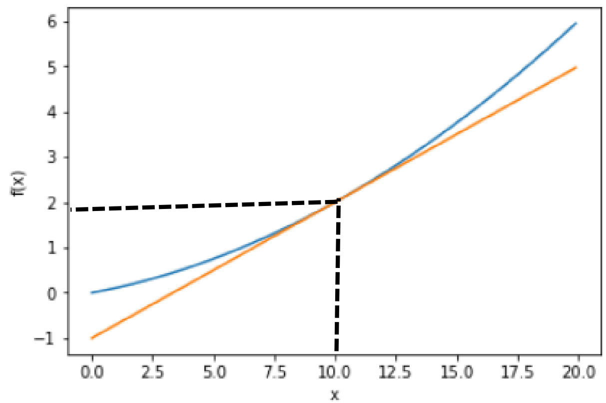
- $x = 10$ 일때 함수의 미분을 계산
  - x에 대한 $f(x)$의 변화량
  - 기울기

```
In [24]: numerical_differential(function_1, 10)
Out[24]: 0.2999999999986347
```

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

```
def tangent_line(f, x):
    d = numerical_diff(f, x)
    print(d)
    y = f(x) - d*x
    return lambda t: d*t + y
```

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

# 0에서 20까지 0.1 간격의 배열 x를 만든다.
x = np.arange(0.0, 20.0, 0.1)
y = function_1(x)

tf = tangent_line(function_1, 10)
y2 = tf(x)

plt.xlabel('x'); plt.ylabel('f(x)')
plt.plot(x, y)
plt.plot(x, y2)
plt.show()
```



Numerical differentiation

❖ 편미분 　변수가 여러 개인 경우 목표 변수에 대한 미분, 나머지는 상수로 고정 처리

- $f(x_0, x_1) = x_0^2 + x_1^2$를 미분.

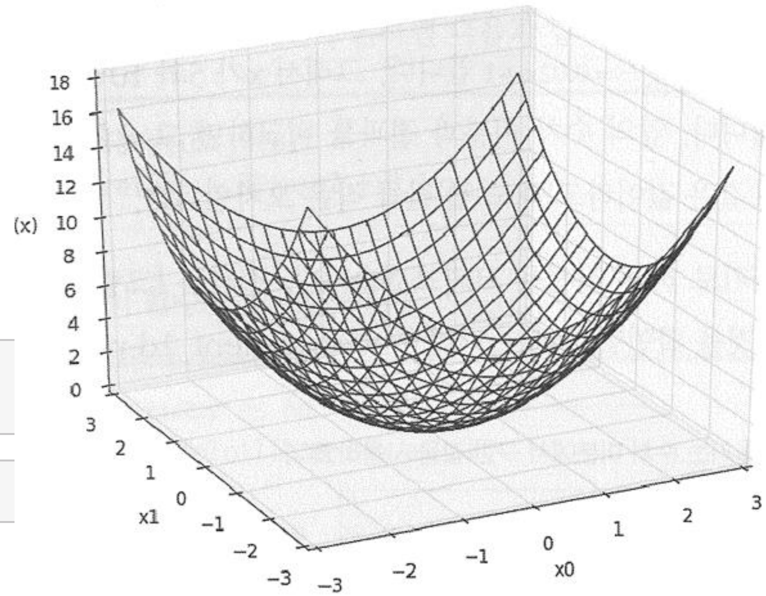
```
In [25]: def function_2(x):
             return x[0]**2 + x[1]**2
```

$x_0 = 3, x_1 = 4$일 때, x_0에 대한 편미분 $\frac{\partial f}{\partial x_0}$를 구하라.

```
In [24]: def function_tmp1(x0):
             return x0*x0 + 4.0**2.0

In [25]: numerical_differential(function_tmp1, 3.0)

Out[25]: 6.00000000000378
```

$x_0 = 3, x_1 = 4$일 때, x_1에 대한 편미분 $\frac{\partial f}{\partial x_1}$를 구하라.

```
In [26]: def function_tmp2(x1):
             return 3.0**2.0 + x1*x1

In [27]: numerical_differential(function_tmp2, 4.0)

Out[27]: 7.999999999999119
```



3

4. Gradient

Gradient

❖ 기울기

- 모든 변수의 편미분을 벡터로 표현
- $f(x_0, x_1) = x_0^2 + x_1^2$의 편미분을 동시에 계산

$$\left(\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1} \right)$$

```python
def _numerical_gradient_no_batch(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] = float(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

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

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

        return grad
```

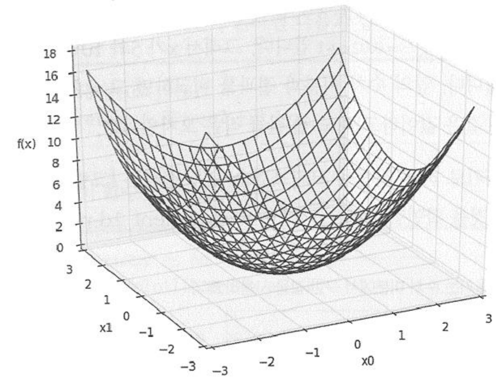
Gradient

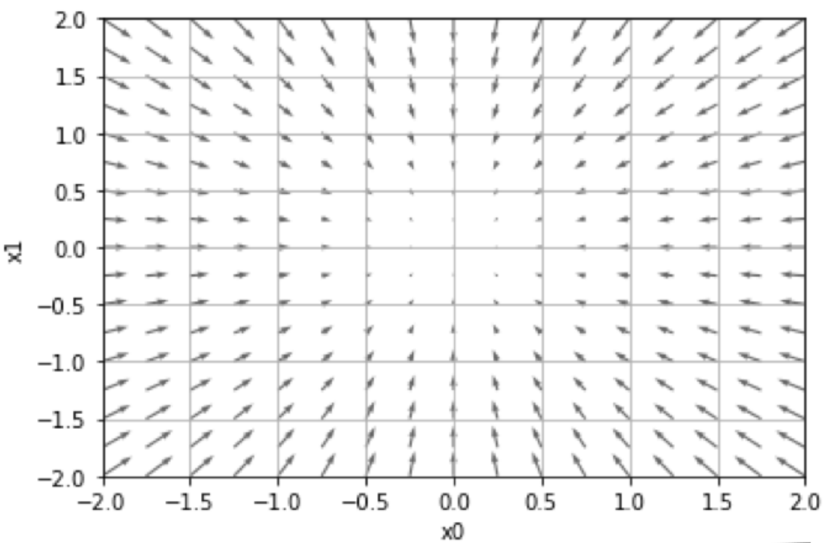
❖ 기울기

- $f(x_0, x_1) = x_0^2 + x_1^2$의 기울기 그래프

```
def function_2(x):
    if x.ndim == 1:
        return np.sum(x**2)
    else:
        return np.sum(x**2, axis=1)
```

```
x0 = np.arange(-2, 2.5, 0.25)
x1 = np.arange(-2, 2.5, 0.25)
X, Y = np.meshgrid(x0, x1)

X = X.flatten()
Y = Y.flatten()

grad = numerical_gradient(function_2, np.array([X, Y]) )

plt.figure()
plt.quiver(X, Y, -grad[0], -grad[1],  angles="xy",color="#666666")
plt.xlim([-2, 2])
plt.ylim([-2, 2])
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.grid()
plt.legend()
plt.draw()
plt.show()
```



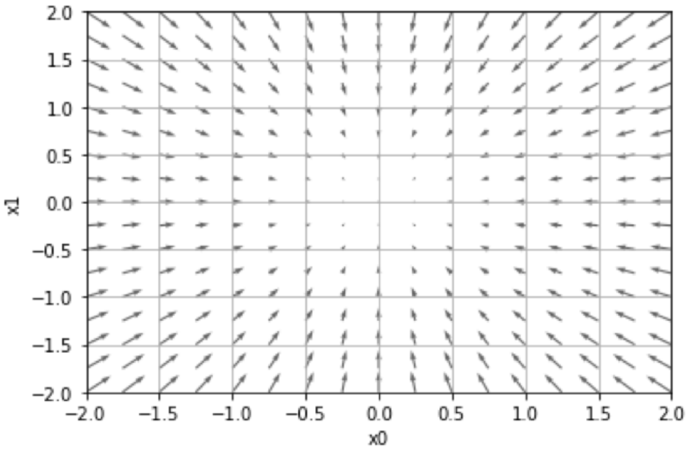


Gradient

❖ 기울기

- (3, 4), (0, 2), (3, 0) 에서의 기울기를 확인.

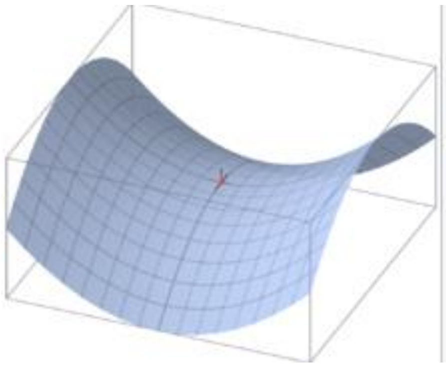
```
In [29]: numerical_gradient(function_2, np.array([3.0, 4.0]))
Out[29]: array([6., 8.])

In [30]: numerical_gradient(function_2, np.array([0.0, 2.0]))
Out[30]: array([0., 4.])

In [31]: numerical_gradient(function_2, np.array([3.0, 0.0]))
Out[31]: array([6., 0.])
```



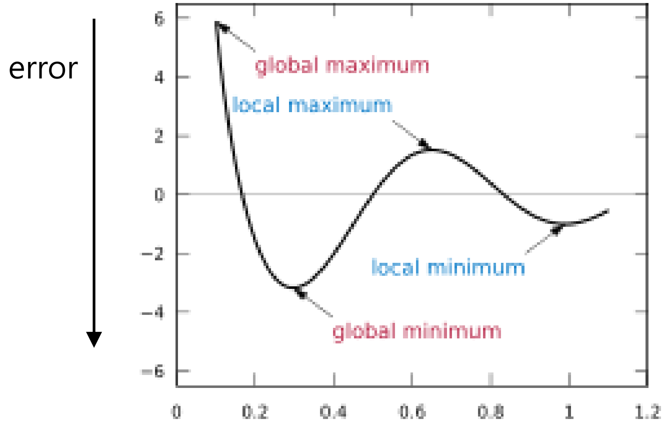
Gradient

❖ Gradient Descent (경사 하강법)

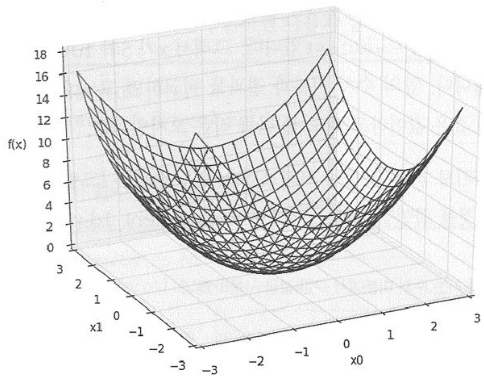
- 최적(optimization): 손실 함수가 최솟값이 될 때의 매개변수
    - 일반적인 문제의 손실 함수는 복잡하고 매개변수 공간이 광대하여 최솟값인지 알아내기 어려움
- 경사하강법: 기울기를 이용하여 함수의 최솟값을 찾으려는 것
    - 함수가 극솟값, 최솟값, 또는 안장점(saddle point)이 되는 장소에서는 기울기가 0
    - 복잡하고 찌그러진 모양의 함수의 경우, 대부분 평평한 곳으로 파고들면서 고원(plateau)이라 하는 학습이 진행되지 않는 정체기에 빠짐
    - 오목함수(Convex function) 일 경우 최소값이 하나임



고원(plateau) 예



Global minimum, Local minimum,



Convex function

Gradient

❖ Gradient Descent (경사 하강법)

- 경사법

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

기존 값에서 학습률과 기울기를 곱한 값을 뺀값으로 갱신

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

- η : 학습률(Learning rate)
  - 한번의 학습으로 얼마만큼 학습해야 할지, 즉 매개변수 값을 얼마나 갱신하는지 정하는 것
  - 학습률이 너무 크거나 작으면 목표하는 곳에 도달하지 못하므로 여러번 번복하면서 서서히 진행

- gradient_descent()
  - f : 최적화 하려는 함수
  - Init_x : 초깃값
  - lr : Learning Rate
  - step_num : 반복 횟수

In [26]:
```
def gradient_descent(f, init_x, lr=0.01, step_num = 100):
    x = init_x
    x_history = []

    for i in range(step_num):
        x_history.append(x.copy())
        grad = numerical_gradient(f, x)
        x -= lr * grad
    return x, np.array(x_history)
```

Gradient

❖ Gradient Descent (경사 하강법)

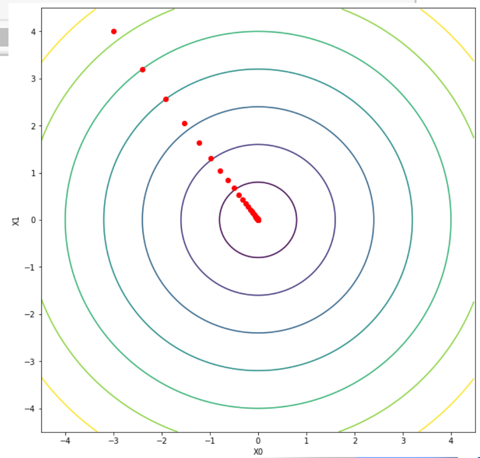
- Quiz

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

```
In [33]: def function_2(x):
             return x[0]**2 + x[1]**2
```

```
In [34]: init_x = np.array([-3.0, 4.0])
         lr = 0.1
         step_num = 100
         x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_nu
         print(x)
```

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



Gradient

❖ 경사법: 학습률이 너무 크거나 작으면 ?

In [40]:
```
# 학습률이 너무 클 때 : lr = 10.0
init_x = np.array([-3.0, 4.0])
x, x_history = gradient_descent(function_2, init_x = init_x, lr = 10.0, ste
print(x)
```

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

In [41]:
```
# 학습률이 너무 작을 때 : lr = 1e-10
init_x = np.array([-3.0, 4.0])
x, x_history = gradient_descent(function_2, init_x = init_x, lr = 1e-10, st
print(x)
```

```
[-2.99999994  3.99999992]
```

- 학습률이 너무 크면 큰 값으로 발산
- 학습률이 너무 작으면 거의 갱신되지 않음

- 하이퍼파라미터(hyper parameter, 초매개변수): 가중치와 편향과는 다르게, 사람이 직접 설정해야 하는 매개변수
  - 여러 후보 값 중에서 시험을 통해 가장 잘 학습하는 값을 찾는 것이 중요

Gradient

❖ 신경망에서의 기울기

- 신경망 학습에서의 기울기: 가중치 매개변수에 관한 손실 함수의 기울기
  - 예) 2X3, 가중치가 **W**, 손실 함수가 ***L***인 신경망의 경사: $\frac{dl}{dW}$

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

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

- $\frac{dl}{dW}$의 각 원소는 각각의 원소에 관한 편미분
  - 예: 1행 1번째 원소인 $\frac{dl}{dW11}$은 w_{11}을 조금 변경했을 때 손실 함수 ***L***이 얼마나 변화하는가를 나타냄

Gradient

❖ 신경망에서의 기울기

- 코드구현

In [44]:
```
class simpleNet:
    def __init__(self):
        self.W = np.random.randn(2,3)

    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
```

Gradient

❖ 신경망에서의 기울기

- 간단한 입력에 대한 구현된 신경망의 예측 결과 확인
- 입력에 대한 정답으로 손실함수 확인

```
In [45]: net = simpleNet()
         print(net.W)

[[ 0.8181476   0.11938052  0.7878456 ]
 [ 0.153049   -1.08694113 -0.27477795]]

In [46]: x = np.array([0.6, 0.9])
         p = net.predict(x)
         print(p)

[ 0.62863266 -0.9066187   0.22540721]

In [47]: np.argmax(p)

Out[47]: 0

In [48]: t = np.array([0, 0, 1])
         net.loss(x, t)

Out[48]: 1.0363903579335143
```

Gradient

❖ 신경망에서의 기울기

- 기울기 확인

```
In [49]: def f(W):
             return net.loss(x, t)

In [50]: dW = numerical_gradient(f, net.W)
         print(dW)

         [[ 0.31854514  0.06861511 -0.38716025]
          [ 0.47781771  0.10292267 -0.58074038]]

In [51]: f = lambda w: net.loss(x, t)
         dW = numerical_gradient(f, net.W)
         print(dW)

         [[ 0.31854514  0.06861511 -0.38716025]
          [ 0.47781771  0.10292267 -0.58074038]]
```

3

5. 학습 알고리즘 구현하기

학습 알고리즘 구현하기

❖ 신경망 학습의 절차

- 전제
  - 신경망에는 가중치와 편향이 존재
  - 학습: 가중치와 편향을 훈련 데이터에 맞게 조정하는 것
- 1 단계: 미니 배치
  - 훈련 데이터 중 일부를 무작위로 가져옵니다. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실 함수 값을 줄이는 것이 목표
- 2 단계: 기울기 산출
  - 미니배치의 손실 함수 값을 줄이기 위해 각 가중치 매개변수의 기울기를 구하며, 기울기는 손실 함수의 값을 가장 적게 하는 방향을 제시
- 3 단계: 매개변수 갱신
  - 가중치 매개변수를 기울기 방향으로 아주 조금 갱신
  - *미니배치를 사용한 경사하강법으로 매개변수 갱신: 확률적 경사 하강법(Stochastic gradient decent)
- 반복
  - 1 ~ 3 단계 반복

학습 알고리즘 구현하기

❖ 2층 신경망 클래스 구현하기

```
import sys, os
sys.path.append("./dataset")
import numpy as np
import pickle
from mnist import load_mnist
import matplotlib.pylab as plt
```

```
def step_function(x):
    return np.array(x > 0, dtype=np.int)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_grad(x):
    return (1.0 - sigmoid(x)) * sigmoid(x)
def relu(x):
    return np.maximum(0, x)
def relu_grad(x):
    grad = np.zeros(x)
    grad[x>=0] = 1
    return grad
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)
    return np.exp(x) / np.sum(np.exp(x))
```

학습 알고리즘 구현하기

❖ 2층 신경망 클래스 구현하기

```python
def mean_squared_error(y, t):
    return 0.5 * np.sum((y-t)**2)
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)
    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
def softmax_loss(X, t):
    y = softmax(X)
    return cross_entropy_error(y, t)
def numerical_gradient(f, x):
    h = 1e-4
    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
        it.iternext()
    return grad
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기

- TwoLayerNet

```
class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        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)
        self.count = 0

    def predict(self, x):
        #print("predict")
        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

    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
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기

- TwoLayerNet

```
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
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기: 클래스 정리

- TwoLayerNet

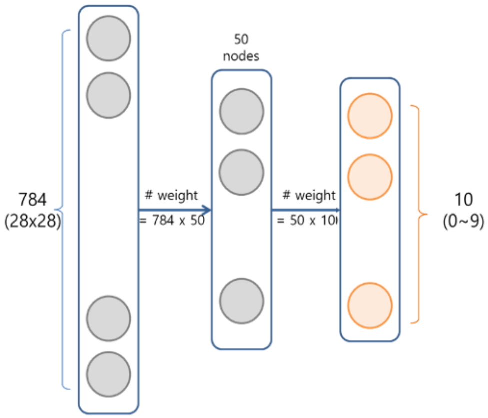
변수		설명
params	1	신경망의 매개변수를 보관하는 딕셔너리 변수 (인스턴스 변수)
	2	params["W1"]은 1번째 층의 가중치, params["b1"]은 1번째 층의 편향
	3	params["W2"]은 2번째 층의 가중치, params["b2"]은 2번째 층의 편향
grads	1	기울기를 보관하는 딕셔너리 변수 (numerical_gradient()메서드의 반환 값)
	2	grads["W1"]은 1번째 층 가중치의 기울기, grads["b1"]은 1번째 층 편향의 기울기
	3	grads["W2"]은 2번째 층 가중치의 기울기,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

메서드		설명
__init__(self, input_size	1	초기화를 수행한다.
, hidden_size, output_size)	1	인수는 순서대로 입력층의 뉴런 수, 출력층의 뉴런 수
predict(self, x)	1	예측(추론)을 수행한다.
	2	인수 x는 이미지 데이터
loss(self, x, t)	1	손실 함수의 값을 구한다.
	2	인수 x는 이미지 데이터, t는 정답 레이블 (아래 3가지 메서드의 인수들도 마찬가지)
accuracy(self, x, t)	1	정확도를 구한다.
numerical_gradient(self, x, t)	1	가중치 매개변수의 기울기를 구한다.
gradient(self, x, t)	1	가중치 매개변수의 기울기를 구한다.
	2	numerical_gradient()의 성능 개선 메서드

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기

- TwoLayerNet



In [60]:
```
net = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
```

In [61]:
```
print(net.params["W1"].shape)
print(net.params["b1"].shape)
print(net.params["W2"].shape)
print(net.params["b2"].shape)
```

```
(784, 50)
(50,)
(50, 10)
(10,)
```

In [62]:
```
x = np.random.rand(100, 784)
y = net.predict(x)
```

In [63]:
```
x = np.random.rand(100, 784)
t = np.random.rand(100, 10)
# 수행시간이 오래 걸림
grads = net.numerical_gradient(x, t)
```

In [64]:
```
print(grads["W1"].shape)
print(grads["b1"].shape)
print(grads["W2"].shape)
print(grads["b2"].shape)
```

```
(784, 50)
(50,)
(50, 10)
(10,)
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기: 미니배치 학습 구현하기

In [66]:
```
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
```

In [67]:
```
train_loss_list = []
```

In [68]:
```
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100 #미니 배치 크기 100
learning_rate = 0.1
iter_per_epoch = max(train_size / batch_size, 1)
```

In [69]:
```
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기: 미니배치 학습 구현하기

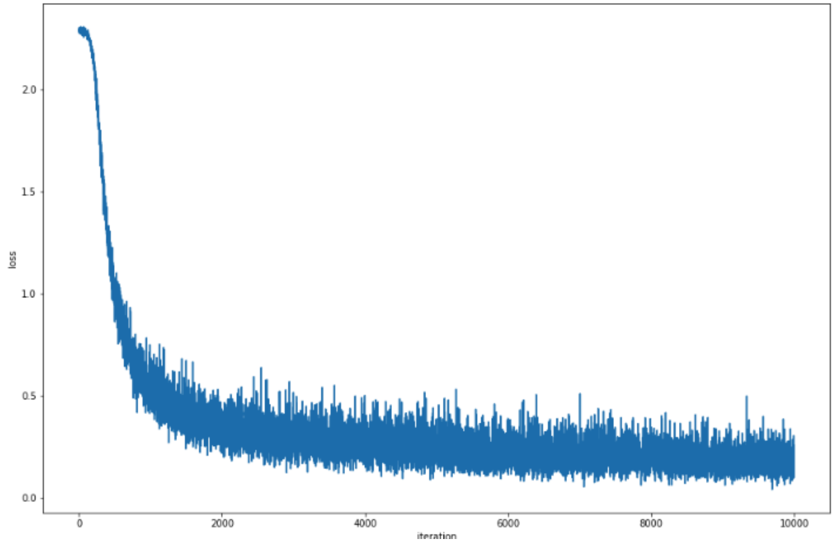
```
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

for step in range(iters_num):
    # Mini-Batch
    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 과정 기록
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)
```

```
loss: 2.279220469854845
loss: 0.9106558228451466
loss: 0.4599994169031179
loss: 0.332367391989282
loss: 0.463966722388639
loss: 0.29392305289214504
loss: 0.16853914953350796
loss: 0.2983127752894409
loss: 0.2238823736477447
loss: 0.23150952605008057
loss: 0.28619094080766344
loss: 0.2612210607685921
loss: 0.27975155533253393
loss: 0.13598944118232534
loss: 0.1897189667810914
loss: 0.127990417548572
loss: 0.23842102041761973
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기: 미니배치 학습 구현하기

```
x = np.arange(len(train_loss_list))

plt.figure(figsize = (15, 10))
plt.plot(x, train_loss_list, label='train acc')
plt.xlabel("iteration")
plt.ylabel("loss")
plt.show()
```



학습 알고리즘 구현하기

❖ 2층 신경망 구현하기: Test Data로 평가하기

```
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]

    # grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch) # 성능 개선판

    # 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)

    # Loss 및 Accuracy 출력
    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("loss: " + str(loss) + ", train acc: " + str(train_acc) + ", test acc: " + str(test_acc))
```

```
loss: 2.2990570415366283, train acc: 0.09751666666666667, test acc: 0.0974
loss: 0.8361739757537845, train acc: 0.8021666666666667, test acc: 0.8037
loss: 0.4069025646993821, train acc: 0.8788166666666667, test acc: 0.8843
loss: 0.3617330967520281, train acc: 0.8982333333333333, test acc: 0.9022
loss: 0.36293519658709356, train acc: 0.9069166666666667, test acc: 0.9093
loss: 0.2132757280939304, train acc: 0.9131666666666667, test acc: 0.917
loss: 0.19104888586236593, train acc: 0.9177166666666666, test acc: 0.9201
loss: 0.2783607150751375, train acc: 0.9220833333333334, test acc: 0.9241
loss: 0.22479098646339796, train acc: 0.9268166666666666, test acc: 0.928
loss: 0.17846401081337754, train acc: 0.9297833333333333, test acc: 0.9306
loss: 0.2563091923451233, train acc: 0.9328166666666666, test acc: 0.9338
loss: 0.26379355418433936, train acc: 0.9351333333333334, test acc: 0.9347
loss: 0.1721236728345652, train acc: 0.9374333333333333, test acc: 0.9364
loss: 0.17561462704535238, train acc: 0.9396666666666667, test acc: 0.9382
loss: 0.2749402299450654, train acc: 0.9414333333333333, test acc: 0.9401
loss: 0.26225419260008587, train acc: 0.94345, test acc: 0.9414
loss: 0.2815445948229342, train acc: 0.9449833333333333, test acc: 0.9437
```

학습 알고리즘 구현하기

❖ 2층 신경망 구현하기: Test Data로 평가하기

```
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))

plt.figure(figsize = (15, 10))
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()
```