4. 신경망 학습

- **학습** : 훈련데이터로부터 가중치 매개변수의 최적값을 자동으로 획득하는 것
- **손실함수** : 신경망이 학습할 수 있도록 해주는 지표
- **학습의목표** : 손실함수의 결과값을 가장 작게 만드는 가중치 매개변수의 값을 찾는 것

4.1 데이터에서 학습한다

- 매개변수의 수가 많고 신경망의 층이 깊어지면 사람의 손으로 매개변수를 정하기 쉽지 않다.
- **퍼셉트론 수렴 정리** : 선형으로 분리 가능한 문제는 유한번의 학습을 통해 풀이가 가능하다는 정리 (비선형 문제의 경우 자동 학습 불가능함)

4.1.1 데이터 주도 학습

이미지를 인식하는 방식

- 이미지 인식 알고리즘을 직접 고안함
    - 이미지안에 숨은 규칙을 명확히 로직으로 풀어내기 쉽지 않은 방법 임
- 기계학습을 이용한 이미지 인식
    - 이미지에서 특징(feature)을 추출하고 그 특빙의 패턴을 기계학습 기술로 학습
    - 이미지의 특징은 보통 벡터로 추출하며 비전분야에서는 SIFT, SURF, HOG 등을 사용함 (사람이만듬)
    - 변환된 벡터를 지도학습 방법인 SVM, KNN 등으로 학습 할 수 있음 (자동화)
- 신경망(딥러닝)방식
    - 사람이 개입하지 않고, 데이터를 있는 그대로 학습 함

4.1.2 훈련데이터와 시험 데이터

- 기계학습 문제는 **훈련데이터**(training data)와 **시험데이터**(test data)로 나눠 학습과 실험을 하는것이 보통임
- 훈련데이터만 사용하여 최적의 매개변수를 찾고 시험데이터로 이 모델의 실력을 평가하는 방법
  (우리가 원하는것은 훈련데이터에 최적화된 모델이 아니라 범용성을 가진 모델인지를 시험하기 위함)
- **오버피팅**(overfitting) : 주어진 데이터 셋에만 지나치게 최적화된 상태를 뜻하는 용어

4.2 손실함수

- 손실함수는 신경망의 성능이 얼마나 **나쁜지**를 측정하는 지표
- 신경망이 훈련데이터를 얼마나 잘 처리하지 **못하느냐**를 나타냄

4.2.1 평균제곱 오차 (Mean Squared Error, MSE)

- 정답에 가까울 수록 평균제곱오차가 작은 값을 가짐
- y_k : 신경망의 출력. (신경망이 추정한 값)

- t_k : 정답 레이블. 정답에 해당하는 차원에만 값이 1이고 나머지는 0을 가짐
    - **원-핫 인코딩** : 한 원소만 1로 하고 나머지 원소를 0으로 나타내는 표기법
- k : 데이터의 차원의 수 (결과값이 가질 수 있는 분류의 갯수 정도로 이해함)

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

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

4.2.2 교차 엔트로피 오차 (Cross Entropy Error, CEE)

- MSE와 함께 손실함수로 많이 사용 됨.
- 자연로그 함수를 사용하므로 log 안의 값이 0이되어 마이너스 무한대의 값이 나올 수 있음
    - 코드로 구현시 로그 안에 아주 작은 값(delta)를 더해서 무한대 값이 나오지 안도록 함
    - 손실함수를 x축의 음의방향으로 평행이동한 것과 같은 효과로
      출력값의 상대적 오차를 구하는데 영향을 주지는 않음

$$E = -\sum_k t_k \log y_k$$

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

4.2.3 미니배치 학습

- 데이터가 모두 N 개라면 교차엘트로피 오차는 다음과 같이 표시 할 수 있음
- t_{nk}는 n 번째 데이터의 k번째 값을 의미, y_{nk}는 신경망의 출력, t_{nk}는 정답레이블을 뜻함

$$E = -\frac{1}{N}\sum_n \sum_k t_{nk} \log y_{nk}$$

- 모든 데이터를 전부 학습하기에는 데이터 양이 많음.
- **미니배치**(mini-batch) : 전체 데이터 중에 일부만 임의로 골라서 학습을 수행하는 방법

4.2.4 (배치용) 교차 엔트로피 오차 구현하기

~~작성 필요함~~

4.2.5 왜 손실함수를 설정하는가?

- 신경망 학습에는 최적의 매개변수(가중치/편향)를 탐색할 때 손실함수의 값이 가능한 작아지는 매개변수값을

찾는다.

- 이 때, 매개변수의 변화에따른 미분값(기울기)을 구하고, 이 미분값을 단서로 매개변수를 서서히 갱신하는 과정을 반복한다.
- 가중치 매개변수의 **손실함수의 미분**이란 **가중치 매개변수의 값을 조금 변화시켯을때, 손실함수가 어떻게 변하는가**의 의미이다.
- 따라서 이 미분값이 0인 경우 손실함수가 가장 작은 값을 가질 것이며, 매개변수의 갱신을 멈춘다.

- 신경망을 학습할 때 **정확도**를 지표로 삼으면 대부분의 장소에서 **미분값이 0**이 된다.
  즉 학습이 진행되는 **방향을 찾을 수 없다.**
  - 정확도는 입력값 데이터 중 올바로 처리한 것의 결과를 나타내며, 매개변수의 변화에 따라 연속적으로 변화하지 않는다. 즉, 해석가능한 함수가 아니라 매개변수의 변화에 따른 **미분**값을 활용 할 수 없다.
  - 이는 계단함수를 활성화함수로 사용하지 않는 이유와도 같다. 계단함수는 대부분의 지점에서 기울기가 0이라 계단함수의 결과를 손실함소의 지표로 사용하는 것은 의미가 없다.

4.3 수치 미분

- **경사법** 에서는 기울기(경사) 값을 기준으로 나아갈 방향을 정한다.

4.3.1 미분

- 미분은 순간의 변화량을 의미하며 다음과 같이 수식으로 표현 할 수 있다.

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

- 이 때, 수치계산중 오차를 줄이기 위해 (x-h) 일 때와 (x+h) 일때의 값을 이용하는 **중앙차분**을 사용 할 수 있다.

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

```
def numerical_diff(f, x):
    h = 1e-4 # 작은 값. 보통 0.0001정도면 적당 함
    return (f(x+h) - f(x-h)) / (2*h)
```

4.3.2 수치미분의 예

- 일 례로, $y = 0.01x^2 + 0.1x$ 의 해석적 해는 $\frac{df(x)}{dx} = 0.02x + 0.1$이다.
- x=5 일 때와 x=10일때의 해석적 해는 0.2와 0.3 이지만, 수치미분으로 구한 값은
  이 값과 정확히 일치하지 않고, 약간의 오차를 가지는 것을 확인 할 수 있다.

4.3.3 편미분

- 다음은 변수가 2개인 다변수 함수의 예 이다.

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

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

- 변수가 여럿인 함수에 대한 미분을 편미분 이라 한다.
- 다음은 $x_0 = 3, x_1 = 4$일 때 $\frac{\partial f}{\partial x_0}$ 를 구하는 코드의 예 이다. (x_1을 상수로 취급)

```
def function_tmp1(x0):
    return x0*x0 + 4.0**2

numerical_diff(function_tmp1, 3.0)
```

4.4 기울기

- **기울기**(gradient) : 모든 변수의 편미분을 벡터로 정리한 것
- 기울게 벡터가 가리키는 곳은 **각 장소에서 함수의 출력값을 가장 크게 줄이는 방향**을 의미 함

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

```
def numerical_gradient(f, x):
    h = 1e-4   #0.0001
    grad = np.zeros_line(x) #x와 형상이 같은 배열 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)
        # gradient 계산
        grad[idx] = (fxh1 - fxh2)/(2*h)
        x[idx] = tmp_val #값 복원

    return grad
```

4.4.1 경사하강법

- 신경망 학습에서도 기계학습과 마찬가지로 학습단계에서 **최적의 매개변수**(가중치/편향)를 찾아야 함
    - 최적 : 손실함수가 최솟값이 될 때의 매개변수의 값
- 매개변수의 공간이 넓어 어디가 최소값이 되는 지점인지 단번에 찾기 어려우며,
  이런 상황에서 **기울기**를 이용해 함수의 최솟값을 찾으려는 것이 **경사하강법** 이다.
    - 기울기가 가리키는 방향이 항상 최솟값이 존재하는 방향은 아니다.

(안장점 saddle point, 고원 platesu 등의 형태인 경우가 반례)
    - 하지만 기울기를 따라 움직이면 함수의 값을 줄일 수 있는 것은 맞다.
    - 수치적으로 접근하면, **경사법**은 현 위치에서 기울기가 가리키는 방향으로 일정 거리만큼 움직이는 작업을
      반복적으로 수행하여 함수의 값을 점차 줄여나가는 과정을 의미한다.
    - 이 기울기를 따라 움직이다보면 극소값을 구할 수 있고, 잘하면 최소값을 구할 수도 있다.
      (참고 : 극소값 중 제일 작은 값이 최소값, http://blog.naver.com/at3650/40141157306)

- 경사하강법을 수식으로 나타내면 아래와 같다.
    - η 는 갱신하는 양을 뜻하며, 이를 신경망 학습에서는 **학습률**(learning rate) 라고 부른다.
    - 아래 식은 1에 해당하는 갱신을 표현한 식이며, 이를 반복하여 함수의 값을 줄인다.
    - 학습율(η) 는 0.01 이나 0.001 등 작은 크기의 특정 값으로 미리정해야 한다.

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$
$$x_1 = x_1 - \eta \frac{\partial f}{\partial x_1}$$

- 이를 코드로 나타내면 다음과 같다
    - f : 최적화하려는 함수
    - init_x : 초기값
    - lr : 학습률(learning rate) , step_num : 경사법에 따른 반복 횟수

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

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

    return x
```

- **하이퍼파라메터**(hyper parameter) : 학습률 과 같이 사전에 사람이 지정해 줘야 하는 매개변수 값
    - 학습률을 너무 크거나 너무 작은 값으로 설정하면 값이 거의 갱신되지 않거나 발산해 버리는 경우가 있다.
    - $f(x_0, x_1) = x_0^2 + x_1^2$ 의 식에서 시작점 (-3.0, 4.0) 일때,
      학습률이 10.0이면 발산하고, 10^{-10}이면 거의 갱신안됨 (교재 pp.133 코드 참고)

4.4.2 신경망 에서의 기울기

- 형상이 2x3인 가중치 **W**와 손실함수가 **L**인 신경망의 기울기는 $\frac{\partial L}{\partial W}$로 나타낼 수 있다.
    - $\frac{\partial L}{\partial W}$의 각 원소는 각각의 원소에 대한 편미분으로 가중치 **W** 와 형상이 같다.
    - 이 때 $\frac{\partial L}{\partial w_{11}}$은 1행1열의 원소인 w_{11}을 조금 변경했을 때 손실함수의 변화를 의미한다.

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

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

- 다음은 간단한 신경망의 예 이다.
  - SimpleNet 라는 클래스를 선언했으며 형상이 2x3인 가중치 배열을 가지고 있다.
  - 이 신경망의 활성화함수는 소프트맥스 함수이며, 오차함수는 교차엔트로피오차(CEE)를 사용한다.

```
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


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
```

- 신경망의 기울기는 다음과 같은 과정으로 구해 볼 수 있다.
  1. 신경망의 입력 x 와 이 입력데이터 처리결과의 정답인 t 를 선언한다.
  2. 신경망 클래스를 초기화한다. 이때 신경망의 가중치 W는 `net.w`에 선언되어 있다.
  3. 손실함수를 선언한다. 코드에서의 f 는 앞의 수식의 **L** 을 의미한다.
     일반 함수 선언방식을 사용해도 되지만, 람다식으로 선언하면 아래 코드와 같이 간편하게 정의할 수 있다.
  4. 손실함수의 기울기를 계산한다. 코드에서의 dW는 앞의 수식의 $\frac{\partial L}{\partial W}$를 의미한다.

```
# 1. 입력값 x와 정답레이블 t 선언
x = np.array([0.6, 0.9])
t = np.array([0, 0, 1])

# 2. 신경망 클래스를 초기화한다.
net = simpleNet()

## 3. 손실함수를 선언한다.
f = lambda w: net.loss(x, t)

## 4. 손실함수의 미분인 기울기(gradient를 계산한다.)
dW = numerical_gradient(f, net.W)
print(dW)
```

- 위 코드의 **dW**가 다음과 같이 출력됐을때를 분석 해 보자. (실제 코드 실행시마다 값은 변한다.)
  - $\frac{\partial L}{\partial w_{11}}$은 0.21이다. 이는 w_{11}을 h 만큼 늘리면 솔실함수는 0.21h 만큼 증가함을 뜻한다.
  - $\frac{\partial L}{\partial w_{32}}$ 는 -0.54이다. 이는 w_{32}을 h 만큼 늘리면 솔실함수는 0.54h 만큼 감소함을 뜻한다.
  - 따라서 손실함수를 줄이려면 w_{11}은 음의방향으로, w_{32}는 양의방향으로 갱신해야 함을 의미하며,
  - 한번에 갱신되는 양에는 w_{11}보다 w_{32}가 기야하는 비중이 더 크다는 것을 알 수 있음

```
[[ 0.21    0.14   -0.36 ]
 [ 0.32    0.22   -0.54 ]]
```

4.5 학습 알고리즘 구현하기

신경망 학습의 절차

- 신경망에는 적응 가능한 **가중치**와 **편향**이 있고, 이 가중치와 편향을 조절하는 것을 **학습**이라고 한다.
  1. **미니배치**
     훈련데이터중 일부를 무작위로 선별해서 가져 온 것.
     이 데이터의 **손실함수**의 값을 줄이는 것이 학습의 목표임
  2. **기울기 산출**
     미니배치의 손실함수의 값을 줄이기위해 기울기를 구함.
     기울기는 손실함수의 값을 가장 작게 하는 방향을 제시함
  3. **매개변수 갱신**
     가중치 매개변수를 기울기의 방향으로 아주 조금 갱신함.
     갱신하는 양을 **학습률**이라고 함.
  4. **반복**
     1~3 단계 반복
- 미니배치 데이터를 경사하강법으로 매개변수의 값을 줄여가는 방법을 **확률적 경사 하강법**이라고 함
- 확률적 경사하강법(Stochastic Gradient Descent)의 머리글자를 따 **SGD**라고 부르기도 함

4.5.1 2층 신경망 클래스 구현하기

- 아래는 2층 신경망을 하나의 클래스로 구현 한 코드임

    - 2층 신경망 이므로 가중치도 $W^{(1)}, W^{(2)}$ 2개, 편향도 $b^{(1)}, b^{(2)}$ 의 2개를 설정한다.
    - 가중치와 편향은 **params**라는 딕셔너리형 변수에 저장하며,
      가중치와 편향의 수치미분 경과도 같은 모양의 배열데이터로 **grads**라는 변수에 저장한다.
    - 생성자의 파라메터로는 입력층의 뉴런의 수(input_size), 은닉층의 뉴런의 수(hidden_size), 출력층의 뉴런의 수(output_size)와 편향의 기준 값(weight_init_std) 으로 구성되어 있다.
    - input_size는 입력 데이터의 수에 따라 결정되며, output_size은 출력 데이터의 수에 따라 결정되지만, 은닉층의 뉴런의 갯수인 hidden_size는 사용자가 임의로 설정할 수 있다.
- 뉴런의 추론을 실행하는 함수는 **predict**이다.
    - 이때 철번째 은닉층에서의 활성화함수는 시그모이드를,
    - 두번째 은닉층의 활성화함수는 소프트맥스 함수를 사용하고 있으며,
    - 출력층의 활성화함수는 항등함수로 코드상에서는 생략되어 있다.
- 추론결과를 평가하는 오차함수는 **loss**이며 교차엔트로피오차(CSE)를 이용한다.
- **accuracy**함수는 추론결과로 나온 답과 실제 접답테이블을 비교하여 추론결과의 정답율을 계산한다.
- **numerical_gradient**함수는 은닉층의 가중치와 편향의 수치미분을 계산해 딕셔너리타입으로 반환한다.

```
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
```

4.5.2 미니배치 학습 구현하기

- 다음은 경사법으로 MNIST데이터의 미니배치를 학습하는 예시이다.
  - 미니배치의 크기를 100으로 하고,
  - 경사법에 의한 반복 횟수를 10000번으로 설정해여
  - 매 반복마다 경사하강법을 이용하여 매개변수를 갱신한다.
- 매 학습시마다 오차를 계산하여 train_loss_list 배열에 저장한다.
  - 학습횟수가 진행 될 수록 오차가 줄어드는 것을 볼 수 있음
  - 즉, 데이터를 반복 학습함으로서 매개변수가 데이터에 적응하고 있다는 것을 뜻함

```
import sys, os
sys.path.append(os.pardir)  # 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)

# 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 = []
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)

    # 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)
```

4.5.3 시험데이터로 평가하기

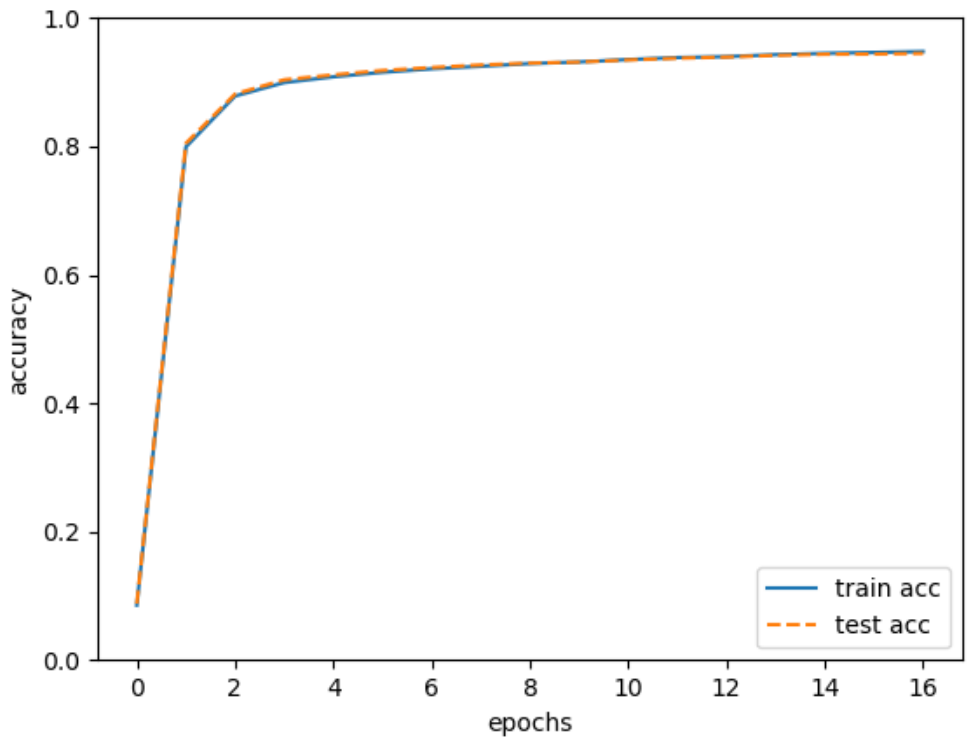
- 학습 데이터를 이용하여 구한 가중치 값이 일반적인 경우에까지 적용 가능한지 검증이 필요함
    - 특정한 학습데이터에만 최적화된 오버피팅이 일어나지 않았는지 확인해야 함
    - 오버피팅이 일어난경우 학습데이터는 제대로 처리하지만 그 이외의 데이터는 잘 처리하지 못할 수 있음
- **에폭**(Epoch) : 학습시 훈련데이터를 모두 소진했을때의 횟수를 기준으로 하는 단위
    - 예를들어 훈련데이터 10000개를 100개 크기의 미니배치로 학습할 경우,
    - 확률적경사하강법으로 100회 반복하여 학습했을 때,
    - 모든 훈련데이터를 소진했다고 보고 이를 1 에폭 이라고 부른다.

- 다음은 앞서 2차 신경망 클래스를 이용한 MNIST학습을 진행하여
  1에폭마다 훈련데이터와 시험데이터의 정확도를 계산하고 그래프로 표시한 예 이다.

```
import sys, os
sys.path.append(os.pardir)  # 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()
```

- 위 코드의 실행결과로 출력된 그래프는 아래와 같다.
  - 훈련데이터에대한 정확도는 실선으로 , 시험데이터에 대한 정확도는 점선으로 표시함
  - 에폭이 진행 될 수록(학습이 진행 될 수록) 훈련데이터와 시험데이터의 정확도가 좋아지고있음
  - 두 정확도 사이에 차이가 거의 없는것으로 보아 이 학습에는 오버피팅이 일어나고 있지 않음



- 만약 오버피팅이 일어난다면??
  - 훈련이란 훈련데이터의 정확도를 높이는 방향으로 진행되므로
    훈련데이터의 정확도는 점점높아지는것이 일반적임
  - 가중치가 훈련데이터에 지나치게 적응하기 시작하면 시험데이터의 정확도가 떨어지기 시작함
    이 지점이 오버피팅이 일어나기 시작하는 지점임

- 오버피팅이 시작되는 순간을 포착해 학습을 중단하면 오버피팅을 효과적으로 예방할 수 있음
  이러한 방법을 **조기종료(early stopping)**라 하며 6장에서 추가로 학습할 예정