밑바닥부터 시작하는 딥러닝 CHATER 4

신경망 학습

데이터 주도 학습

기계학습의 중심에는 데이터가 존재한다. 이미지에서 특징을 추출하고 그 특징의 패턴을 기계학습 기술로 학습하는 방법이 있다. 여기서의 특징은 입력데이터(입력 이미지)에서 본질적인 데이터(중요한 데이터)를 정확하게 추출할 수 있도록 설계된 변환기를 가리킨다. 이미지의 특징은 보통 벡터로 기술하고, 컴퓨터 비전 분야에서는 SIFT, SURF, HOG 등의 특징을 많이 사용한다. 이런 특징을 사용하여 이미지 데이터를 벡터로 변환하고, 변환된 벡터를 가지고 지도 학습 방식의 대표 분류 기법인 SVM, KNN 등으로 학습할 수 있다. 이와 같은 기계학습에서는 모아진 데이터로부터 규칙을 찾아내는 역할을 기계가 담당한다.

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

신경망은 이미지를 있는 그대로 학습한다. 두 번째 접근 방식(특징과 기계학습 방식)에서는 특징을 사람이 설계했지만, 신경망은 이미지에 포함된 중요한 특징까지도 기계가 스스로 학습할 것이다.

신경망의 이점은 모든 문제를 같은 맥락에서 풀 수 있다는 점에 있다. 세부사항과 관계없이 신경망은 주어진 데이터를 온전히 학습하고, 주어진 문제의 패턴을 발견하려 시도한다. 즉 신경망은 모든 문제를 주어진 데이터 그대로를 입력 데이터로 활용해 end-to-end로 학습할 수 있다.

훈련데이터와 시험 데이터

기계학습 문제는 데이터를 훈련데이터와 시험데이터로 나눠 학습과 실험을 수행하는 것이 일반적이다. 우선 훈련데이터만 사용하여 학습하면서 최적의 매개변수를 찾는다. 그 다음 시험 데이터를 사용하여 앞서 훈련한 모델의 실력을 평가하는 것이다. 이렇게 하는 이유는 원하는 것을 범용적으로 사용할 수 있는 모델이기 때문이다. 이 범용 능력을 제대로 평가하기 위해 훈련 데이터와 시험 데이터를 분리하는 것이다. 범용능력은 아직 보지 못한 데이터로도 문제를 올바르게 풀어내는 능력이다. 데이터셋 하나로만 매개변수의 학습과 평가를 수행하면 올바른 평가가 될 수 없다. 한 데이터셋에만 지나치게 최적화된 상태를 overfitting이라고 한다.

손실 함수

손실함수는 신경망 성능의 나쁨을 나타내는 지표로 현재의 신경망이 훈련 데이터를 얼마나 잘 처리하지 못하느냐를 나타낸다.

오차 제곱합

가장 많이 쓰이는 손실 함수는 오차제곱합이다.

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

y는 신경망의 출력, t는 정답 레이블, k는 데이터의 차원 수를 나타낸다.

오차제곱합을 구현하면 다음과 같다.

```
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] # 한 원소를 1 나머지는 0으로 하는 것을 원-핫 인코딩이라고 한다.


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

print(sum_squares_error(np.array(y),np.array(t)))
```

교차 엔트로피 오차

또 다른 손실 함수로써 교차 엔트로피 오차도 자주 이용한다. 교차 엔트로피 오차의 수식은 다음과 같다.

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

y는 신경망의 출력, t는 정답 레이블, t는 정답에 해당하는 인덱스의 원소만 1이고 나머지는 0이다. (원-핫 인코딩). 그렇기 때문에 실질적으로 정답일 때의 추정의 자연로그를 계산하는 식이 된다.

이 교차 엔트로피의 오차를 구현하면 다음과 같다.

```
import numpy as np


t = [0,0,1,0,0,0,0,0,0,0]
y = [0.1,0.05,0.6,0.0,0.05,0.1,0.0,0.1,0.0,0.0]
z = [0.1,0.05,0.1,0.0,0.05,0.1,0.0,0.6,0.0,0.0]

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

print(cross_entropy_error(np.array(y),np.array(t)))
print(cross_entropy_error(np.array(z),np.array(t)))
```

np.log를 계산할 때 delta를 더했는데 이것은 np.log() 함수에 0을 입력하면 -inf가 되어서 계산을 진행할 수 없게 되기 때문이다. 절대 0이 되지 않도록 하기 위해서 delta를 더해주는 것이다.

미니배치 학습

기계학습 문제는 훈련 데이터에 대한 손실 함수의 값을 구하고, 그 값을 최대한 줄여주는 매개변수를 찾아내는데 이렇게 하기 위해선 모든 훈련 데이터를 대상으로 손실 함수 값을 구해야 한다. 훈련 데이터 모두에 대한 손실 함수의 합을 구하는 방법을 생각해볼 때 교차 엔트로피의 오차는 다음과 같이 구해볼 수 있다.

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

데이터가 N개라면 t는 n번째 데이터의 k번째 값을 의미한다. 마지막에 N으로 나누어 정규화하고 있다. N으로 나눔으로써 평균 손실 함수를 구하는 것이다.

하지만 빅데이터 수준이 되면 N이 매우 거대한 값이 되므로 데이터 중 일부를 추려 전체의 근사치로 이용한다. 이러한 일부를 미니배치 라고 한다. 그리고 이것들을 학습시키는 것을 미니배치 학습이라고 한다. 그것을 코드로 구현하면 다음과 같다.

```
import sys,os

sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist # MNIST 데이터 셋을 읽어
오는 함수(훈련 데이터와 시험 데이터를 읽는다)

(x_train, t_train), (x_test, t_test) = \
    load_mnist(normalize=True, one_hot_label=True) # one_ho
t_label = True 로 원-핫 인코딩을 한다. -> 정답위치만 1 나머지 0

print(x_train.shape) # (60000,784)
print(t_train.shape) # (60000,10)

train_size = x_train.shape[0]
batch_size = 10
batch_mask = np.random.choice(train_size, batch_size) # 훈련
데이터 중 10개만 랜덤으로 추출하는 함수
x_batch = x_train[batch_mask]
t_batch = t_train[batch_mask]

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

교차 엔트로피 오차 구현하기

코드로 나타내면 다음과 같다

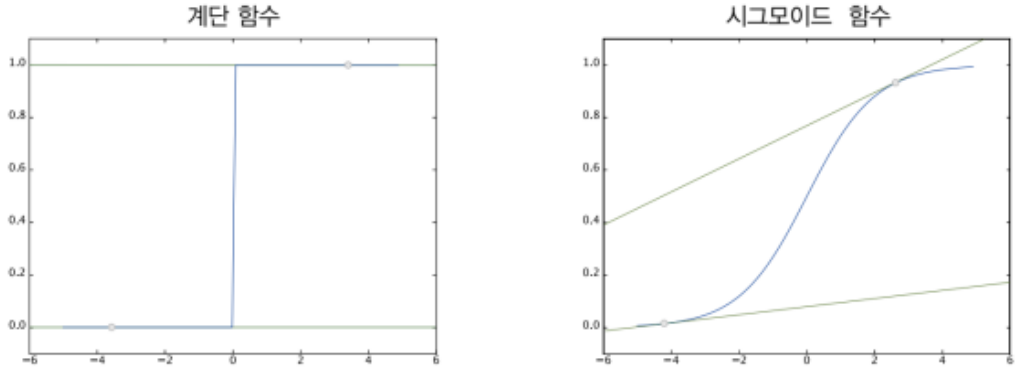
```
def cross_entropy_error(y,t):
    if y.ndim == 1:
        t = t.reshape(1,t.size) # 정답 레이블
        y = y.reshape(1,y.size) # 신경망의 출력

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

y가 1차원이라면, 즉 데이터 하나당 교차 엔트로피 오차를 구하는 경우는 reshape 함수로 데이터의 형상을 바꿔주고 배치의 크기로 나눠서 정규화하고 이미지 1장당 평균의 교차 엔트로피 오차를 계산한다.

손실 함수를 설정하는 이유

신경망 학습에서는 최적의 매개변수를 탐색할 때 손실함수의 값을 가능한 작게 하는 매개변수 값을 찾는다. 이 때 매개변수의 미분을 계산하고 그 미분 값을 단서로 매개변수의 값을 서서히 갱신하는 과정을 반복한다. 가상의 신경망이 있고 그 신경망의 어느 한 가중치 매개변수에 주목한다고 할 때 가중치 매개변수의 손실 함수의 미분이란 가중치 매개변수의 값을 아주 조금 변화시켰을 때, 손실 함수가 어떻게 변하나 라는 의미이다. 만약 이 미분 값이 음수면 그 가중치 매개변수를 양의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있다. 반대로, 미분 값이 양수면 가중치 매개변수를 음의 방향으로 변화시켜 손실 함수의 값을 줄일 수 있다. 만약 미분 값이 0이면 어느 쪽으로 움직여도 손실 함수의 값이 줄어들지 않으므로 가중치 매개변수의 갱신이 멈추게 된다. 정확도를 지표로 삼아서는 안 되는 이유는 미분 값이 대부분의 장소에서 0이 되어 매개변수를 갱신할 수 없기 때문이다.

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



두 그래프를 비교하면 왜 정확도 대신 손실함수를 사용하는지 이해할 수 있다.

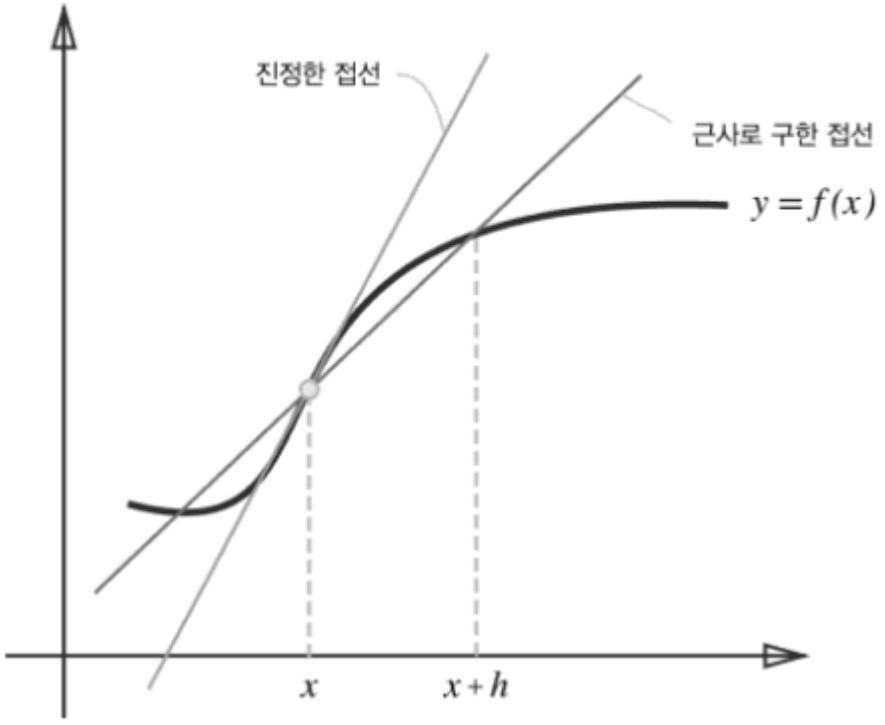
미분

미분은 한순간의 변화량을 표시한 것으로 수식으로 나타내면 다음과 같다.

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

하지만 이 미분계산에는 오차가 있다. 이 오차를 개선한 그래프를 그려보면 다음과 같다.

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



이 오차를 줄이기 위해 (x+h)와(x-h)일 때의 함수 f의 차분을 계산하는 방법을 쓰기도 한다. 이 차분은 x를 중심으로 그 전후의 차분을 계산한다는 의미에서 중심 차분 혹은 중앙 차분이라 한다. 이를 활용하여 수치 미분을 다시 구현하면 다음과 같이 작성할 수 있다.

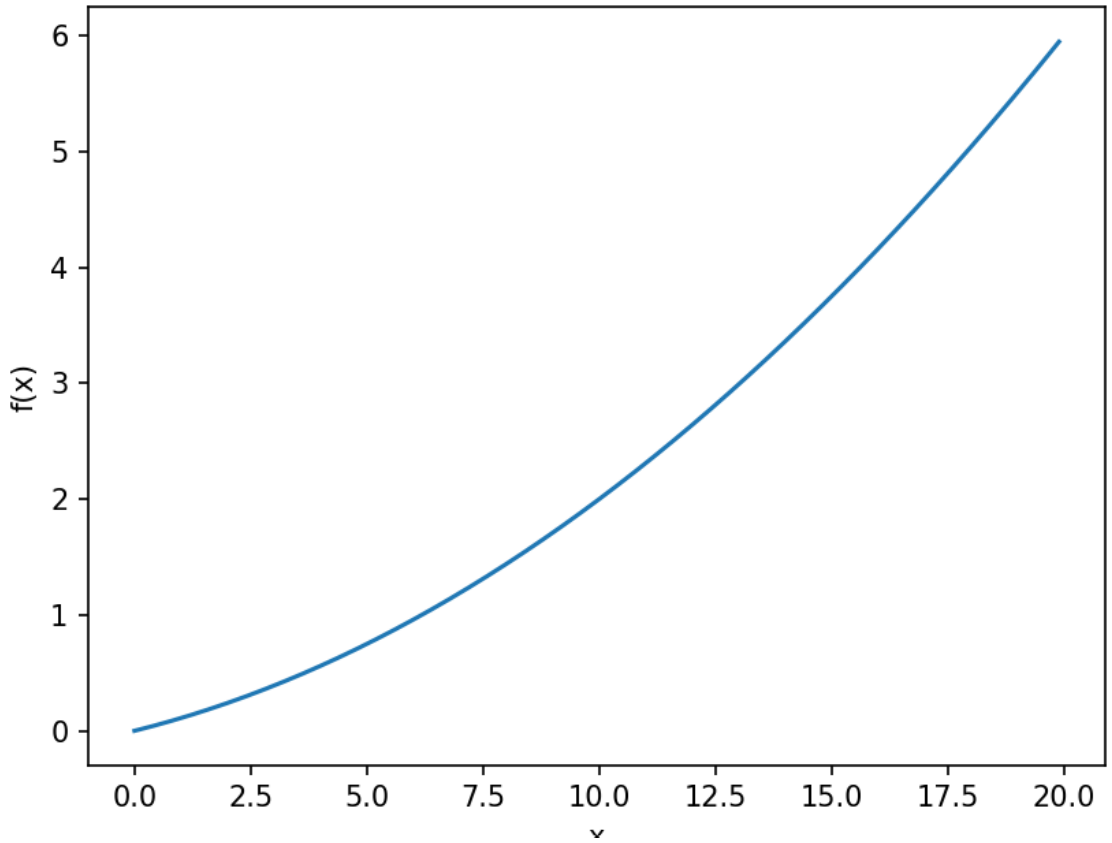
```
def numerical_diff(f,x):
    h = 1e - 4
    return (f(x+h)-f(x-h)) / (2*h)
```

수치 미분의 예

간단한 이차함수를 미분해보면 코드는 다음과 같다.

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

def numerical_diff(f,x):
    h = 0.0001
    return (f(x+h)-f(x-h)) / (2*h)

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

x = np.arange(0.0,20.0,0.1)
y = function_1(x)
plt.xlabel("x")
plt.ylabel("f(x)")
plt.plot(x,y)
plt.show()
```

위 코드를 실행시켜 나온 그래프는



편미분

인수들의 제곱 합을 계산하는 식을 코드로 구현하면 다음과 같다

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

이 때 변수가 2개 이상이므로 이 함수에 대한 미분을 편미분이라고 한다.

편미분은 변수가 하나인 미분과 마찬가지로 특정 장소의 기울기를 구한다. 단 여러 변수 중 목표 변수 하나에 초점을 맞추고 다른 변수는 값을 고정한다.

기울기

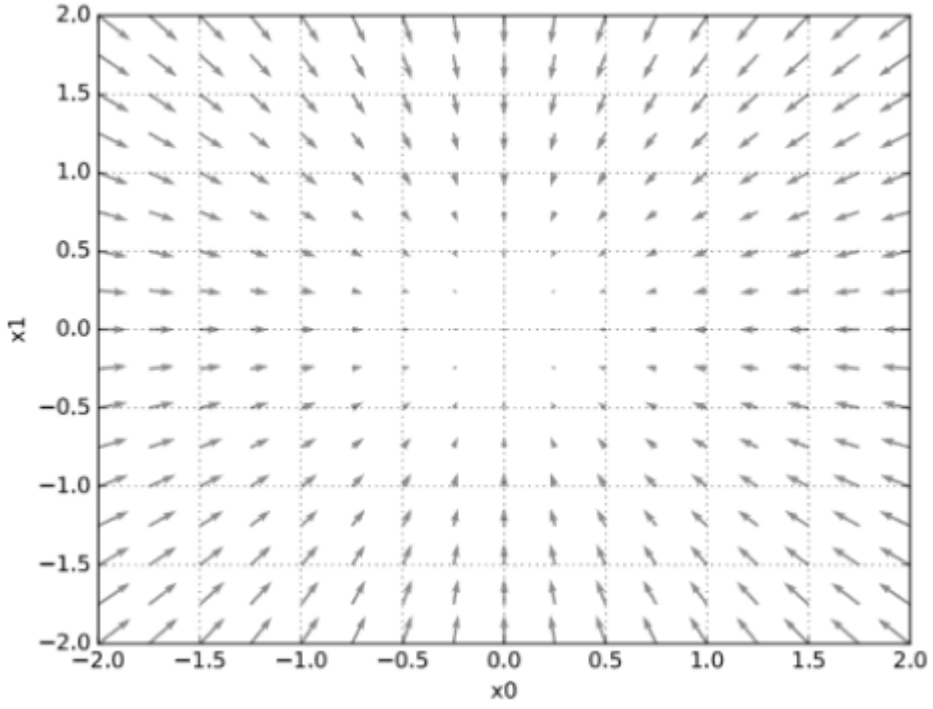
모든 변수의 편미분을 벡터로 정리한 것을 기울기라고 한다. 코드로 구현하면 다음과 같다.

```
def numerical_gradient(f,x):
    h = 0.0001
    grad = np.zeros_like(x) # x와 형상이 같은 배열 형성

    for idx in range(x.size):
```

```
        tmp_val = x[idx]
        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
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])))
```

기울기의 결과에 마이너스를 붙인 벡터를 그려보면 다음과 같다.

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



기울기는 각 지점에서 낮아지는 방향을 가리킨다. **기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향**이라고 볼 수 있다.

경사법(경사 하강법)

기계학습 문제 대부분은 학습 단계에서 최적의 매개변수를 찾아낸다. 여기서 최적이란 손실 함수가 최소값이 될 때의 매개변수 값이다. 기울기를 잘 이용해서 함수의 최소값을 찾으려는 것이 경사법이다.

경사법은 현 위치에서 기울어진 방향으로 일정 거리만큼 이동한다. 그 다음 이동한 곳에서도 마찬가지로 기울기를 구하고, 또 그 기울어진 방향으로 나아가기를 반복한다. 이렇게 해서 함수의 값을 점차 줄이는 것이 경사법이다. 경사법을 수식으로 나타내면 다음과 같다.

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

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

η는 갱신하는 양을 나타낸다. 이를 신경망 학습에서는 학습률이라고 한다. 한 번의 학습으로 얼마만큼 학습해야 할지, 매개변수 값을 얼마나 갱신하느냐를 정하는 것이 학습률이다. 변수의 수가 늘어도 같은 식으로 갱신한다. 또한 학습률의 값은 미리 0.01, 0.01 등 특정 값으로 정해두어야 한다. 신경망 학습에서는 보통 이 학습률 값을 변경하면서 올바르게 학습하고 있는지를 확인하면서 진행한다. 경사하강법을 구현하면 다음과 같다.

```
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
```

f는 최적화하려는 함수, init_x는 초기값, lr는 learning rate, step_num은 경사법에 따른 반복 횟수를 뜻한다. 함수의 기울기는 numerical_gradient(f,x)로 구하고 그 기울기에 학습률을 곱한 값으로 갱신하는 처리를 step_num번 반복한다.

학습률이 너무 크면 큰 값으로 발산하고 반대로 너무 작으면 거의 갱신되지 않은 채 끝나버리므로 학습률을 적절히 설정하는 것이 중요하다.

학습률 같은 매개변수를 하이퍼파라미터라고 한다. 이는 가중치와 편향 같은 신경망의 매개변수와는 성질이 다른 매개변수이다. 신경망의 가중치 매개변수는 훈련 데이터와 학습 알고리즘에 의해 자동으로 획득되는 매개변수인 반면, 학습률 같은 하이퍼파라미터는 사람이 직접 설정해야 하는 매개변수인 것이다. 일반적으로 이 하이퍼파라미터들은 여러 후보 값 중에서 시험을 통해 가장 잘 학습하는 값을 찾는 과정을 거쳐야 한다.

신경망에서의 기울기

신경망 학습에서도 기울기를 구해야 하는데, 여기서의 기울기란 가중치 매개변수에 대한 손실 함수의 기울기이다.

```
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

x = np.array([0.6, 0.9])
t = np.array([0, 0, 1])

net = simpleNet()

f = lambda w: net.loss(x, t)
dW = numerical_gradient(f, net.W)

print(dW)
```

예측을 수행하는 predict(x)와 손실 함수의 값을 구하는 loss(x,t)가 있다.

인수 x는 입력 데이터, t는 정답 레이블이다.

기울기는 numerical_gradient(f,x)를 통해 구할 수 있다. f는 함수 x는 함수 f의 인수이다. 그래서 여기에서는 net,W를 인수로 받아 손실 함수를 계산하는 새로운 함수 f를 정의했다. 그리고 이 새롭게 정의한 함수를 numerical_gradient(f,x)에 넘긴다.

dW는 numerical_gradient(f,net,W)의 결과로, 그 형상은 2x3의 2차원 배열이다. 신경망의 기울기를 구한 다음에는 경사법에 따라 가중치 매개변수를 갱신하기만 하면 된다.

학습 알고리즘 구현하기

신경망 학습의 절차는 다음과 같다.

전체

신경망에는 적용 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다. 4단계로 수행이 된다.

1단계-미니배치

훈련데이터 중 일부를 무작위로 가져온다. 이렇게 선별한 데이터를 미니배치라 하며, 그 미니배치의 손실 함수 값을 줄이는 것이 목표이다.

2단계-기울기 산출

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

3단계-매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신한다.

4단계-반복

1~3단계를 반복한다

이것이 신경망 학습이 이뤄지는 순서이며 경사 하강법으로 매개변수를 갱신하는 방법이다. 이때 데이터를 미니배치로 무작위로 선정하기 떄문에 확률적 경사 하강법이라고 부른다. 대부분의 딥러닝 프레임워크는 확률적 경사 하강법의 영어 머리글자를 딴 SGD라는 함수로 이 기능을 구현하고 있다.

2층 신경망 클래스 구현하기

```
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_siz
e, weight_init_std=0.01):
        # 가중치 초기화
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.ran
dn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.ran
dn(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.param
s['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.param
s['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.param
s['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.param
s['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
```

중요 변수와 메서드를 정리한 표는 다음과 같다.

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...

params 변수에는 이 신경망에 필요한 매개변수가 모두 저장되고 params 변수에 저장된 가중치 매개변수가 예측 처리(순방향 처리)에서 사용된다.

grads 변수에는 params 변수에 대응하는 각 매개변수의 기울기가 저장된다.

TwoLayerNet의 메서드들을 살펴보면 **init** 을 통해 클래스를 초기화한다. 인수는 순서대로 입력층의 뉴런 수, 은닉층의 뉴런 수, 출력층의 뉴런 수이다. 여기에서 가중치 매개변수도 초기화한다. loss(self,x,t)는 손실 함수의 값을 계산하는 메서드이다. 이 메서드에서는 predict()의 결과와 정답 레이블을 바탕으로 교차 엔트로피 오차를 구하도록 구현했다. 남은 numerical_gradient(self,x,t) 메서드는 각 매개변수의 기울기를 계산한다. 마지막 gradient(self,x,t)는 오차역전파법을 사용하여 기울기를 효율적이고 빠르게 계산한다.

미니배치 학습 구현하기

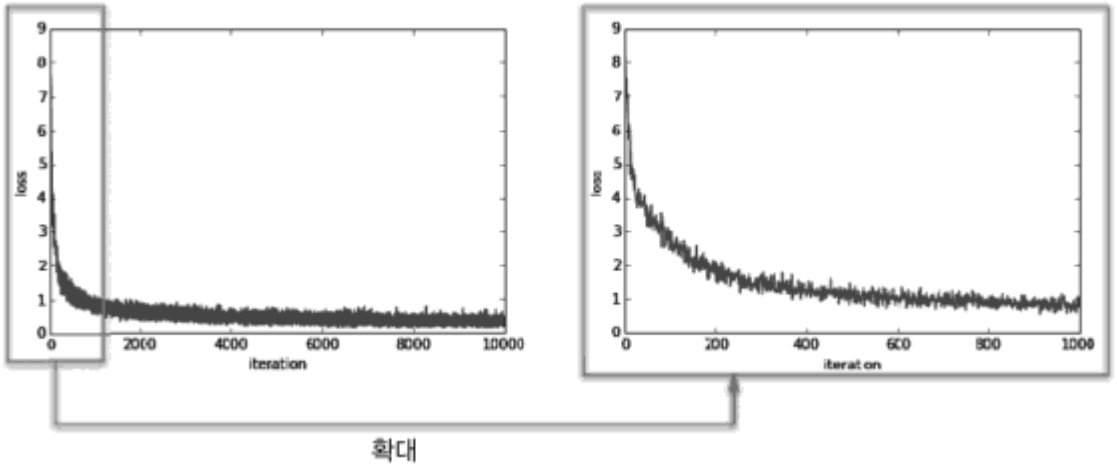
코드로 구현하면 다음과 같다.

```
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

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

train_loss_list=[]

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

network = TwoLayerNet(input_size=784, hidden_size=50, outpu
t_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) # 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)
```

이 코드에서는 미니배치 크기를 100으로 했다. 매번 60,000개의 훈련 데이터에서 임의로 100개의 데이터를 추려내고 그 미니배치를 대상으로 확률적 경사 하강법을 수행해 매개변수를 갱신한다. 경사법에 의한 갱신 횟수를 10,000번으로 설정하고, 갱신할 때마다 훈련 데이터에 대한 손실 함수를 계산하고, 그 값을 배열에 추가한다. 이 손실함수의 값이 변화하는 추이를 그래프로 나타내면 다음과 같다.

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



학습 횟수가 늘어가면서 손실 함수의 값이 줄어든다. 데이터를 반복해서 학습함으로써 최적 가중치 매개변수로 가고 있다는 뜻이 된다.

시험 데이터로 평가하기

위 그래프를 통해 손실 함수의 값이 점점 내려가는 것을 확인했는데 이 손실 함수 값이란 훈련 데이터의 미니배치에 대한 손실 함수의 값이다. 다만 이 결과만으로는 다른 데이터셋에도 비슷한 학습효과를 발휘할 지는 확신할 수 없다.

그래서 훈련 데이터 외의 데이터를 올바르게 인식하는지 확인해야 한다. 즉 overfitting을 일으키지 않는지 확인해야 한다. 이를 위해 다음 구현에서는 학습 도중 정기적으로 훈련 데이

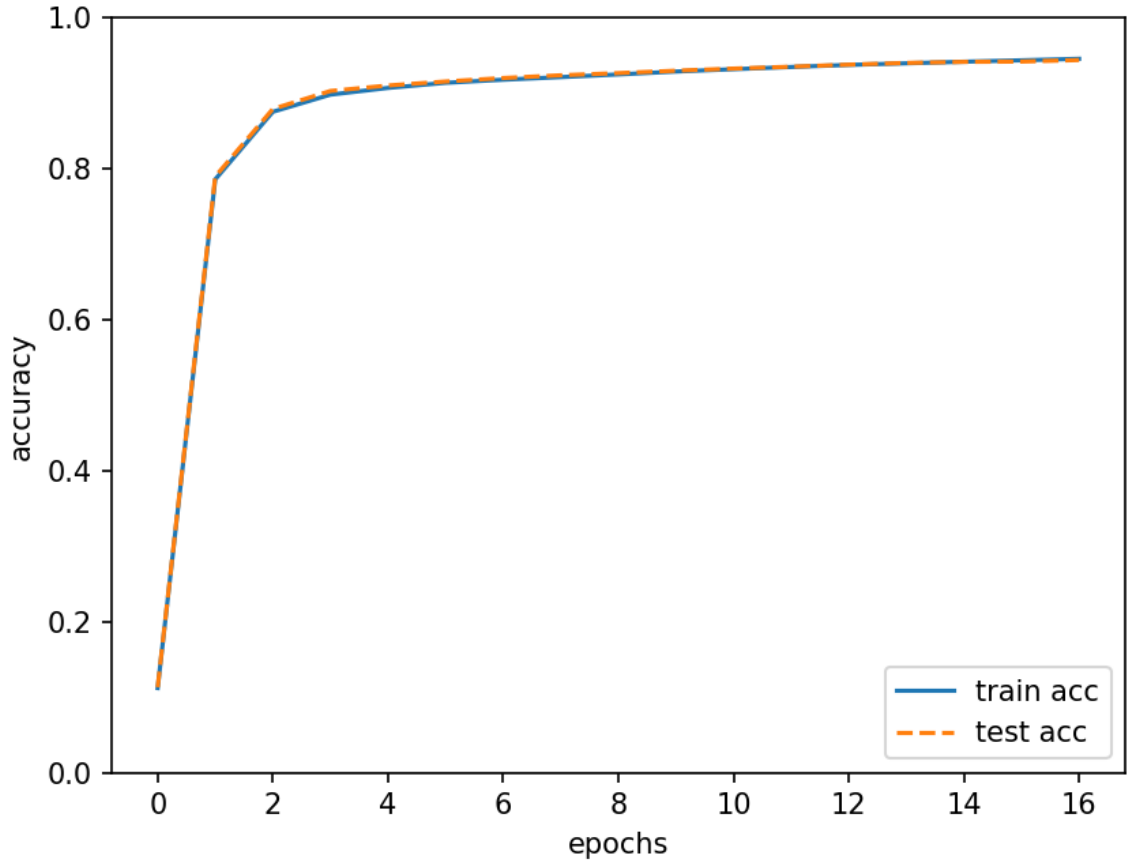
터와 시험 데이터를 대상으로 정확도를 기록한다.

```
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있
도록 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, outpu
t_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()
```

epoch는 하나의 단위이다. 1epoch는 학습에서 훈련 데이터를 모두 소진했을 때의 횟수에 해당한다. 예를 들어 10,000개를 100개의 미니배치로 학습할 경우, 확률적 경사 하강법을 100회 반복하면 모든 훈련 데이터를 소진한 것이 되므로 100회가 1epoch가 된다.

이 코드를 실행시켜 얻은 그래프는 다음과 같다.



학습이 진행될수록 훈련데이터와 시험 데이터를 사용하고 평가한 정확도가 모두 좋아지고 있다.