CHAPTER 5 오차역전파법

4장에서 경사하강법을 이용한 신경망 학습을 공부했는데,
이 때, 수치 미분을 이용해 매개변수의 기울기를 구했습니다.
수치 미분은 간단하고 구현하기 쉽지만 시간이 오래 걸린다는 게 단점입니다.

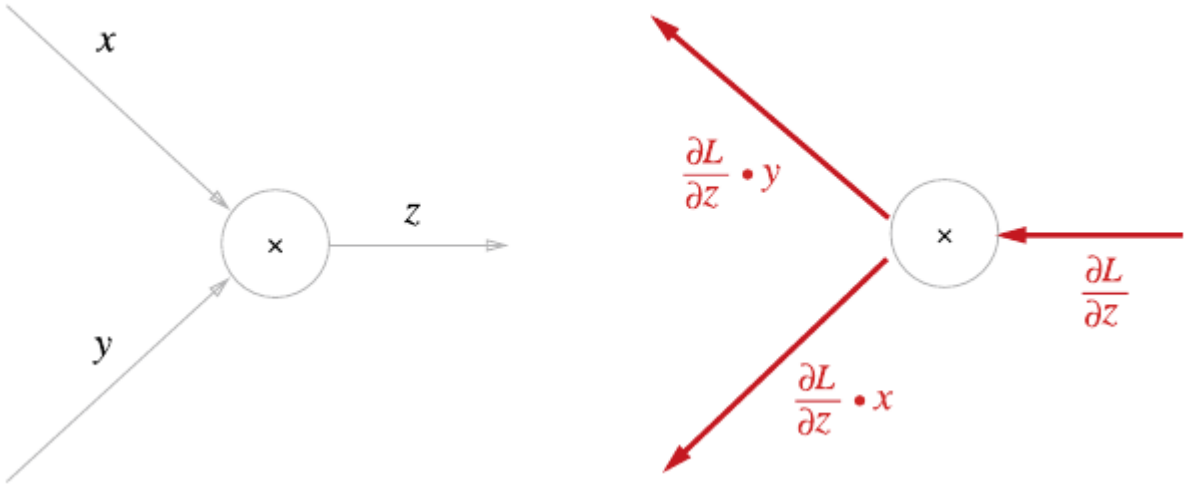
5장에서는, 가중치 매개변수의 기울기를 효율적으로 계산하는 오차역전파법(backpropagation)을 공부하겠습니다.
수식을 이용해서 오차역전파법을 이해할 수 있지만, 우리는 계산 그래프를 이용해서 '시각적 ' 으로 이해하려 합니다.

Contents

5.4.1 곱셈 계층

```
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    def forward(self, x, y):
        self.x = x
        self.y = y
        out = x * y

        return out

    def backward(self, dout):
        dx = dout * self.y # x와 y를 바꾼다.
        dy = dout * self.x

        return dx, dy
```

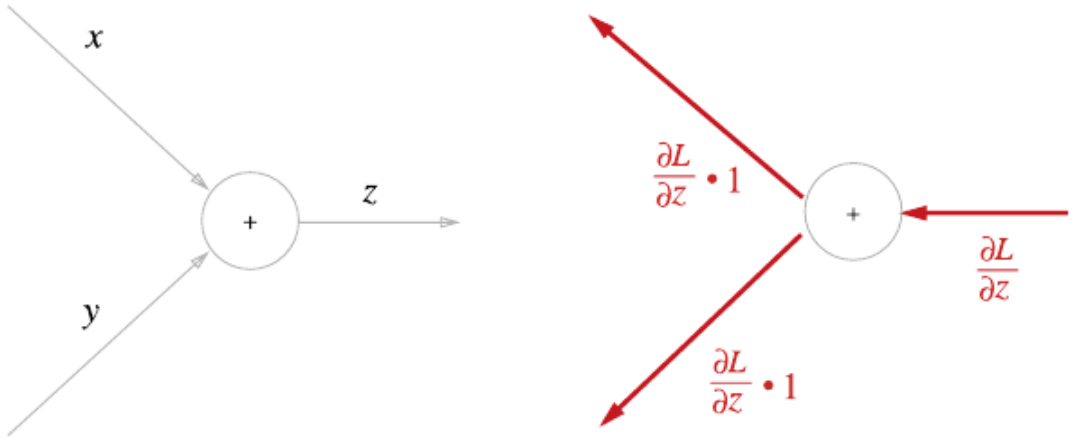
그림 5-12 곱셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다.



5.4.2 덧셈 계층

```
class AddLayer:
    def __init__(self):
        pass

    def forward(self, x, y):
        out = x + y
        return out

    def backward(self, dout):
        dx = dout * 1
        dy = dout * 1
        return dx, dy
```

그림 5-9 덧셈 노드의 역전파 : 왼쪽이 순전파, 오른쪽이 역전파다. 덧셈 노드의 역전파는 입력 값을 그대로 흘려보낸다.



5.5.1 ReLU 계층

```
class Relu:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0

        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout

        return dx
```

common/layers.py

그림 5-18 ReLU 계층의 계산 그래프

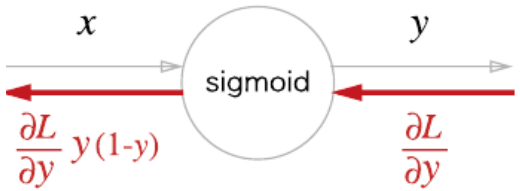
Relu 클래스는 mask라는 인스턴스 변수를 가진다. mask는 True/False로 구성된 넘파이 배열로, 순전파의 입력인 x의 원소 값이 0 이하인 인덱스는 True, 그 외(0보다 큰 원소)는False로 유지한다.

순전파시, (입력 값은 필요없고) 입력 값의 부호를 저장하고 있다가 역전파시 0을 할당한다.

5.5.2 Sigmoid 계층

```
class Sigmoid:
    def __init__(self):
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out

        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out

        return dx
```

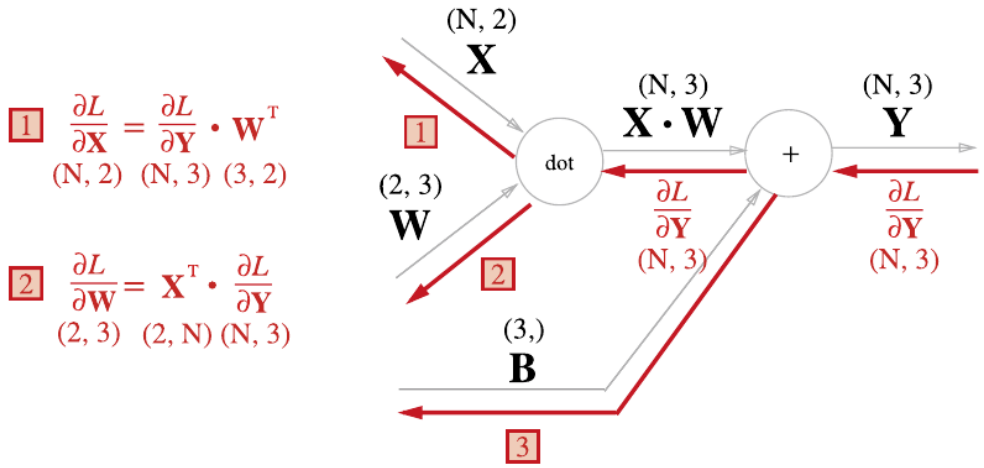
그림 5-22 Sigmoid 계층의 계산 그래프 : 순전파의 출력 y만으로 역전파를 계산할 수 있다.



5.6.2 배치용 Affine 계층

```
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.x = None
        self.dW = None
        self.db = None

    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b

        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)

        return dx
```

그림 5-27 배치용 Affine 계층의 계산 그래프



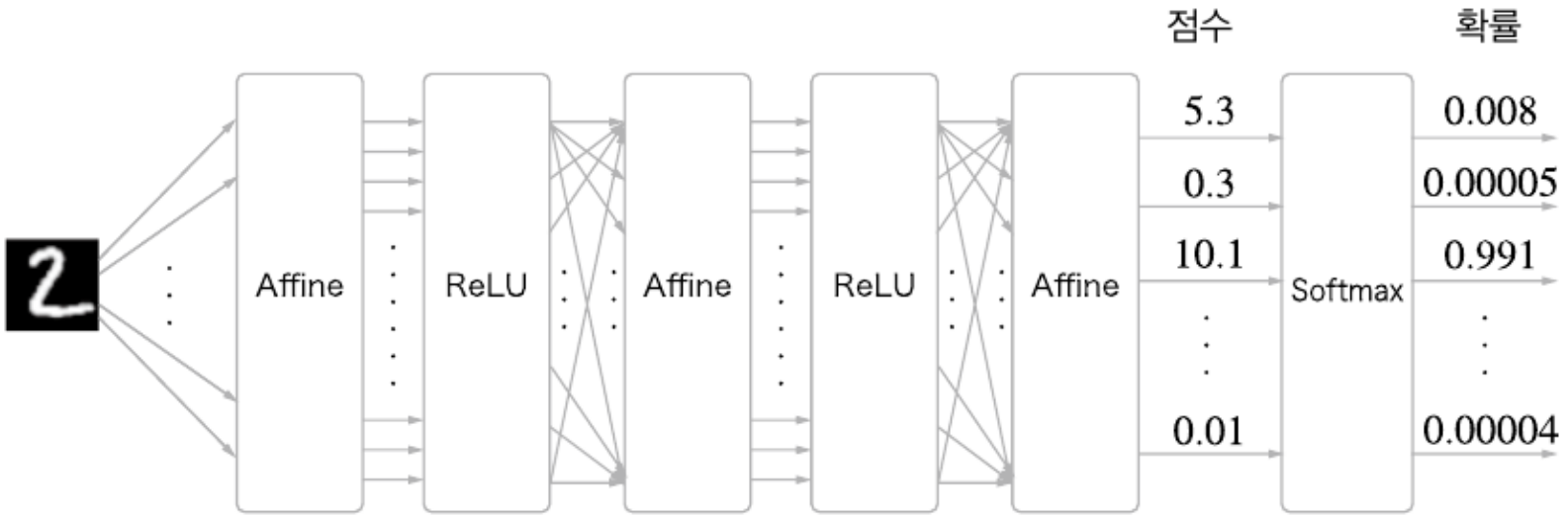
신경망과 구현한 Relu, Sigmoid, Affine, Softmax-with-Loss 계층들과의 관계

복습

5.6.3 Softmax-with-Loss 계층

MNIST 데이터를 위한 3층 신경망 예시

그림 5-28 입력 이미지가 Affine 계층과 ReLU 계층을 통과하며 변환되고, 마지막 Softmax 계층에 의해서 10개의 입력이 정규화된다. 이 그림에서는 숫자 '0'의 점수는 5.3이며, 이것이 Softmax 계층에 의해서 0.008(0.8%)로 변환된다. 또, '2'의 점수는 10.1에서 0.991(99.1%)로 변환된다.



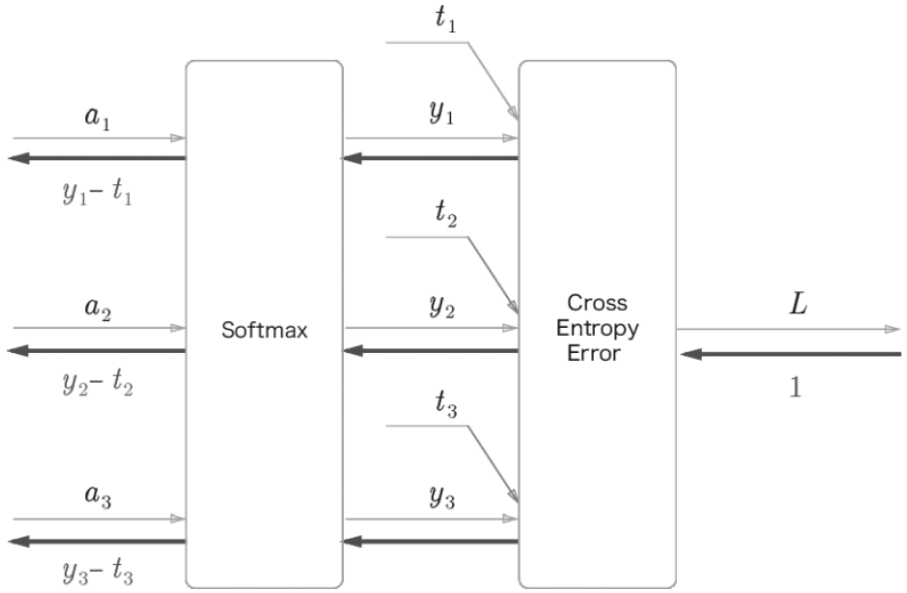
NOTE_ 신경망에서 수행하는 작업은 **학습**과 **추론** 두 가지입니다. 추론할 때는 일반적으로 Softmax 계층을 사용하지 않습니다. 예컨대 [그림 5-28]의 신경망은 추론할 때는 마지막 Affine 계층의 출력을 인식 결과로 이용합니다. 또한, 신경망에서 정규화하지 않는 출력 결과([그림 5-28]에서는 Softmax 앞의 Affine 계층의 출력)를 **점수**score라 합니다. 즉, 신경망 추론에서 답을 하나만 내는 경우에는 가장 높은 점수만 알면 되니 Softmax 계층은 필요 없다는 것이죠. 반면, 신경망을 학습할 때는 Softmax 계층이 필요합니다.

소프트맥스 계층은 입력값을 정규화하여 출력하는 것 뿐이다.

5.6.3 Softmax-with-Loss 계층

```
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None # 손실
        self.y = None    # softmax의 출력
        self.t = None    # 정답 레이블(원-핫 벡터)

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = cross_entropy_error(self.y, self.t)
        return self.loss

    def backward(self, dout=1):
        batch_size = self.t.shape[0]
        dx = (self.y - self.t) / batch_size

        return dx
```

그림 5-30 '간소화한' Softmax-with-Loss 계층의 계산 그래프



복습

5.7.1 신경망 학습의 전체 그림 복습 (4.5절)

전제

신경망에는 적응 가능한 가중치와 편향이 있고, 이 가중치와 편향을 훈련 데이터에 적응하도록 조정하는 과정을 '학습'이라 한다.

신경망 학습은 다음과 같이 4단계로 수행한다.

1단계 – 미니배치 선택

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

2단계 - 기울기 산출

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

3단계 - 매개변수 갱신

가중치 매개변수를 기울기 방향으로 아주 조금 갱신.

4단계 - 반복

1~3단계를 반복.

4.5.1 2층 신경망 클래스 구현하기

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

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

4장에서 구현한

TwoLayerNet 복습(참고용)

5.7.2 오차역전파법을 적용한 신경망 구현하기

계층을 사용해서 다시 구현한

TwoLayerNet

```
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict    # 순서가 있는 딕셔너리, 추가한 순서를 기억함.

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)

        # 계층 생성
        self.layers = OrderedDict()
        self.layers['Affine1'] = Affine(self.params['W1'], self.params['b1'])
        self.layers['Relu1']   = Relu()
        self.layers['Affine2'] = Affine(self.params['W2'], self.params['b2'])

        self.lastLayer = SoftmaxWithLoss()

    # 다음 페이지 계속~
```

```
def predict(self, x):
    for layer in self.layers.values():
        x = layer.forward(x)                    # softmax 안 함
    return x

def loss(self, x, t):                           # x : 입력 데이터, t : 정답 레이블
    y = self.predict(x)
    return self.lastLayer.forward(y, t)

def accuracy(self, x, t):
    y = self.predict(x)
    y = np.argmax(y, axis=1)
    if t.ndim != 1 : t = np.argmax(t, axis=1)

    accuracy = np.sum(y == t) / float(x.shape[0])
    return accuracy

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

이제부터, 느린 numerical_gradient는 안 씀!

(노드가 아닌) '계층'으로 모듈화해서 구현한 효과가 매우 크다. 신경망의 구성요소를 계층으로 구현한 덕분에, 신경망을 쉽게 구축할 수 있다. 만약, 5층, 10층, 20층 등 깊은 신경망을 만들고 싶다면, 단순히 필요한 만큼 더 추가하면 된다. 레고 블록 조립하듯.

```
def gradient(self, x, t):
    # forward
    self.loss(x, t)

    # backward
    dout = 1
    dout = self.lastLayer.backward(dout)

    layers = list(self.layers.values())
    layers.reverse()
    for layer in layers:
        dout = layer.backward(dout)

    # 결과 저장
    grads = {}
    grads['W1'], grads['b1'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
    grads['W2'], grads['b2'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

    return grads
```

이제부터, 빠른 gradient 메서드를 쓴다!

즉, 경사하강법의 그레디언트를 구하기 위해 이 함수를 호출함.

Common.layers 모듈

지금까지 설명한 계층(layer)들을 파이썬 클래스로 구현한 코드 모음.

6장에서 공부하는 신경망 학습을 위한 고급 테크닉들이 계층(layer)으로 구현되어 추가됨.

코드를 유의하여 천천히 읽어 보시면 많은 도움이 됩니다.

https://github.com/WegraLee/deep-learning-from-scratch/blob/master/common/layers.py

5.7.2 오차역전파법을 적용한 신경망 구현하기

5장에서 구현한 계층들을 이용하여, 4장에서 구현했던 TwoLayerNet 클래스를 효과적으로 다시 구현했다.

다시 구현한 TwoLayerNet 클래스의 인스턴트 변수와 메서드는 다음 표와 같다.

표 5-1 TwoLayerNet 클래스의 인스턴스 변수

인스턴스 변수	설명
params	딕셔너리 변수로, 신경망의 매개변수를 보관
	params['W1']은 1번째 층의 가중치, params['b1']은 1번째 층의 편향
	params['W2']는 2번째 층의 가중치, params['b2']는 2번째 층의 편향
layers	순서가 있는 딕셔너리 변수로, 신경망의 계층을 보관
	layers['Affine1'], layers['Relu1'], layers['Affine2']와 같이 각 계층을 순서대로 유지
lastLayer	신경망의 마지막 계층
	이 예에서는 SoftmaxWithLoss 계층

표 5-2 TwoLayerNet 클래스의 메서드

메서드	설명
__init__(self, input_size, hidden_size, output_size, weight_init_std)	초기화를 수행한다.
	인수는 앞에서부터 입력층 뉴런 수, 은닉층 뉴런 수, 출력층 뉴런 수, 가중치 초기화 시 정규분포의 스케일
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
accuracy(self, x, t)	정확도를 구한다.
numerical_gradient(self, x, t)	가중치 매개변수의 기울기를 수치 미분 방식으로 구한다(앞 장과 같음).
gradient(self, x, t)	가중치 매개변수의 기울기를 오차역전파법으로 구한다.

5.7.3 오차역전파법으로 구한 기울기 검증하기

지금까지 기울기를 구하는 방법 두가지를 설명했다. 하나는 수치 미분을 써서 구하는 방법, 또 하나는 해석적으로 수식을 풀어서 구하는 방법.

두 방식으로 구한 기울기가 일치함(엄밀히 말하면 거의 같음)을 확인하는 작업을 기울기 확인gradient check이라고 한다.

```
import numpy as np
from mnist import load_mnist
import time
#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)

x_batch = x_train[:3]
t_batch = t_train[:3]
# 다음 페이지 계속~
```

5.7.3은 두 방법으로 구한 기울기가 같다는 것이 결론이다. 그냥 연습 삼아 이 코드를 타이핑하고 실행해 보기 바랍니다.

5.7.3 오차역전파법으로 구한 기울기 검증하기

```
start_time = time.time()
grad_numerical = network.numerical_gradient(x_batch, t_batch)
time_grad_numerical = round(time.time()-start_time,1)

start_time = time.time()
grad_backprop = network.gradient(x_batch, t_batch)
time_grad_backprop = round(time.time()-start_time,1)

# 각 가중치의 절대 오차의 평균을 구한다.
for key in grad_numerical.keys():
    diff = np.average( np.abs(grad_backprop[key] - grad_numerical[key]) )
    print(key + ":" + str(diff))

print("grad_numerical time:", time_grad_numerical)
print("grad_backprop time:", time_grad_backprop)
```

출력 결과

```
W1:4.3117955332683286e-10
b1:2.6324135385901154e-09
W2:5.908118590390327e-09
b2:1.4026418079515767e-07
grad_numerical time: 10.2
grad_backprop time: 0.0
```

효율(efficiency)이 높다!
(시간이 많이 단축된다)

5.7.4 오차역전파법을 사용한 학습 구현하기

```
(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)

iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []

iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]
    # 다음 페이지 계속~
```

5.7.4 오차역전파법을 사용한 학습 구현하기

```
# 기울기 계산
# grad = network.numerical_gradient(x_batch, t_batch) # 수치 미분 방식
grad = network.gradient(x_batch, t_batch)    # 오차역전파법(훨씬 빠르다): 4장 코드와 이 부분만 다르다

# 갱신
for key in ('W1', 'b1', 'W2', 'b2'):
    network.params[key] -= learning_rate * grad[key]

loss = network.loss(x_batch, t_batch)
train_loss_list.append(loss)

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_list.append(t (train_acc)
    test_accest_acc)
    print(round(train_acc,4), round(test_acc,4))
```

출력 결과

```
0.123 0.1236
0.904 0.9075
0.9247 0.9273
0.9381 0.9359
0.946 0.9437
0.9524 0.9492
0.9574 0.9559
0.9616 0.9587
0.9653 0.9609
0.9679 0.9637
0.9691 0.964
0.9721 0.9673
0.9738 0.9685
0.9742 0.9671
0.976 0.9708
0.9776 0.9687
0.9782 0.9706
```

사람마다 출력 결과가 다를 수 있다. Why?

5.8 정리

이번 장에서 배운 내용

- 계산 그래프를 이용하면 계산 과정을 시각적으로 파악할 수 있다.
- 계산 그래프의 노드는 국소적 계산으로 구성된다. 국소적 계산을 조합해 전체 계산을 구성한다.
- 계산 그래프의 순전파는 통상의 계산을 수행한다. 한편, 계산 그래프의 역전파로는 각 노드의 미분을 구할 수 있다.
- 신경망의 구성 요소를 계층으로 구현하여 기울기를 효율적으로 계산할 수 있다(오차역전파법).
- 수치 미분과 오차역전파법의 결과를 비교하면 오차역전파법의 구현에 잘못이 없는지 확인할 수 있다(기울기 확인).