머신러닝4. 신경망 학습

신경망 학습

신경망 학습에서의 학습

-훈련데이터로부터 가중치 매개변수의 최적값을 자동으로 획득하는 것

손실함수

-신경망이 학습을 하게 해주는 지표

-손실함수의 값이 적을 수록 좋으므로 손실함수의 결과값을 가장 작게 만드는 가중치 매개변수를 찾는 것이 목표

여기서는 그 방법으로 **함수의 기울기**를 활용함

신경망과 딥러닝

-기존 기계학습에서 사용하던 방법보다 사람의 개입을 더욱 배제할 수 있도록함.

즉, 사람이 처음부터 설계하고 규칙을 만들어내서 기계에게 학습시키는 것이 아닌

이미 주어진 데이터를 통해 기계 스스로 규칙을 찾게 한다거나 본질적인 것을 추출할 수 있도록 함.

Ex)강아지 얼굴 구별하는 것이나 손 글씨 판별과 같은 것

-중요한 특징까지 기계가 스스로 학습하는 것

-이 책에서는 이미지를 주로 다루고 있음

*주의할 점-오버피팅 피하기(한 데이터셋에 만 지나치게 최적화된 상태)

손실함수

y_k 신경망이 추정한 값

t_k 정답 레이블

신경망도 하나의 지표를 기준으로 최적의 매개변수 값을 측정하는데 이때의 하나의 지표가 손실함수!
종류 1.오차제곱합
2.교차엔트로피 오차

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

-**오차제곱합**: 값이 작을 수록 정답에 가까운 것

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

-**교차엔트로피오차**: 밑이 e인 자연로그

정답인 경우의 신경망이 도출한 확률값이 작으면 오차는 커짐.

둘 다 값이 작을 수록 신경망이 추정한 값이 정답과 가깝다고 볼 수 있음.

미니배치 학습

-데이터가 너무 많으면 모두 학습할 수 없으니 신경망에서도 데이터로부터 일부만 골라 학습을 수행함.
-일부=미니배치
-무작위로 데이터를 랜덤하게 뽑음

데이터의 개수가 N개인 데이터 여러 개에 대한 교차엔트로피오차

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

정확도가 아닌 손실함수를 이용하는 이유

-신경망 학습에서는 최적의 매개변수를 탐색할 때 손실함수의 값을 작게 하는 매개변수 값을 찾는데 이때 매개변수의 미분(정확히는 기울기)를 계산함

-그러나 정확도를 지표로 하면 매개변수의 미분이 대부분 0이 되기 때문에 정확도를 지표로 삼지 않음.

-대부분 0이 되는 이유: 정확도는 매개변수의 미세한 변화에는 거의 반응을 보이지 않고 반응이 있더라도 연속적이지 않고 갑자기 변화하기 때문

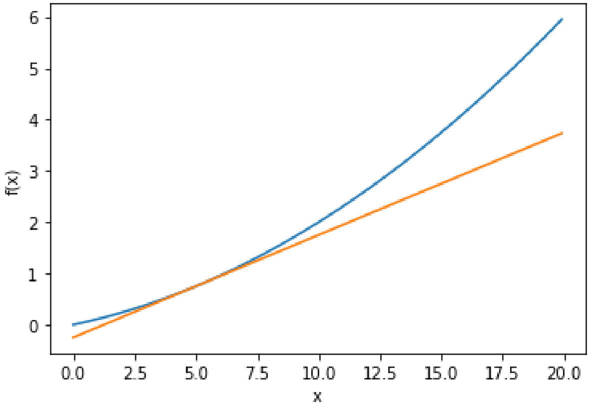
4.3

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

```
# 나쁜 구현 예
def numerical_diff(f, x):
    h = 10e-50
    return (f(x + h) - f(x)) / h
```

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

```
0.1999999999990898
```



미분

-한순간의 변화량

-X의 작은 변화가 함수를 얼만큼 변화시키느냐를 의미

정석식

문제1 :h를 가급적 무한히 0에 가깝게 하고 싶었으나 파이썬에서의 반올림 오차로 인해 너무 작은 값을 사용하면 계산에 문제가 생김

문제2: h를 무한히 0으로 좁힐 수 없기 때문에 진정한 접선을 구할 수가 없음

따라서 해당 식을 사용

(x+h)와 (x-h)일 때의 함수의 차분을 계산.

이렇게 사용하면 진정한 미분과 오차가 매우 적음.

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

문제 1 : $x_0 = 3, x_1 = 4$일 때, x_0에 대한 편미분 $\frac{\partial f}{\partial x_0}$ 를 구하라.

```
>>> def function_tmp1(x0):
...     return x0*x0 + 4.0**2.0
...
>>> numerical_diff(function_tmp1, 3.0)
6.00000000000378
```

문제 2 : $x_0 = 3, x_1 = 4$일 때, x_1에 대한 편미분 $\frac{\partial f}{\partial x_1}$ 를 구하라.

```
>>> def function_tmp2(x1):
...     return 3.0**2.0 + x1*x1
...
>>> numerical_diff(function_tmp2, 4.0)
7.999999999999119
```

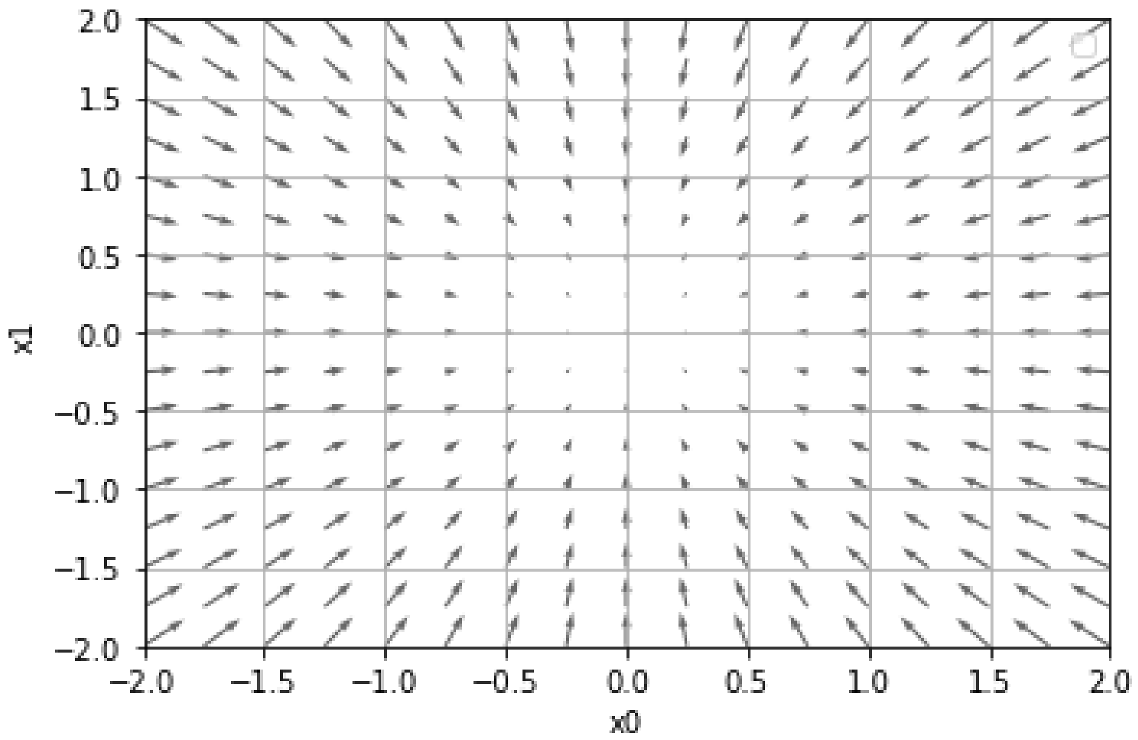
편미분

변수가 여럿인 함수에 대한 미분

변수가 하나인 미분과 마찬가지로 특정장소의 기울기를 구함.

그러나 변수 중 목표 변수 하나에 초점을 맞춰 다른 변수는 값을 고정.

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



기울기의 결과에 마이너스를 붙인 벡터 그림

기울기

편미분을 동시에 계산하고 싶을 때는 양쪽의 편미분을 묶어서 계산함 이때

기울기: 모든 변수의 편미분을 묶어서 정리한 것

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

기울기가 의미하는것

-기울기는 각 지점에서 낮아지는 방향을 가리키고 있음

- <u>기울기가 가리키는 쪽은 각 장소에서 함수의 출력 값을 가장 크게 줄이는 방향</u>

```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
```

기울기 구현방법 코드

```python
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
```

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

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

즉, 손실함수의 값을 낮추는 방안을 제시하는 것이 기울기
기울어진 방향으로 가야 함수의 값을 줄일 수 있음
기울기정보를 단서로 나아갈 방향을 정해야하는 것

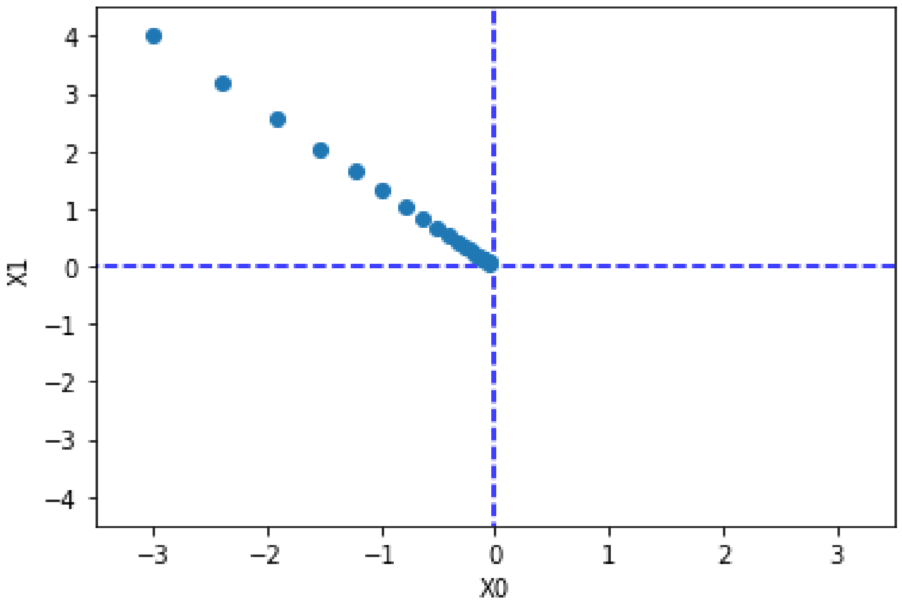
경사하강법
현 위치에서 기울어진 방향으로 일정 거리만큼 이동, 다시 기울기를 구하여 나아가고 반복하는 것.

η

에타: 갱신하는양=학습률=한번의 학습으로 얼만큼 학습해야하는지, 매개변수 값을 얼마나 갱신하는지를 정하는 것

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

```python
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)


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

init_x = np.array([-3.0, 4.0])

lr = 0.1
step_num = 20
x, x_history = gradient_descent(function_2, init_x, lr=lr, step_num=step_num)
```



경사법으로 최솟값 구하는법

-F: 최적화하려는 함수
-Init_x: 초기값
-lr:학습률(보통 0.01이나 0.001등 미리 특정값으로 정해둠) , 학습률 값을 변경해가면서 올바르게 학습되고 있는지 확인함
-Step_num: 경사법반복횟수
-numeral_gradient:함수의 기울기

신경망에서의 기울기

```python
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)
```

```
[[ 0.10529441  0.37394066 -0.47923506]
 [ 0.15794161  0.56091099 -0.71885259]]
```

$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

W:가중치

L:손실함수

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$$

W를 변경했을때 손실함수가 얼마나 변화하느냐

-형상이 2*3인 가중치 매개변수하나를 인스턴스 변수로 가짐
-predict(x): 예측을 수행
-loss(x,t): 손실함수의 값(x는 입력데이터, t는 정답레이블)
-f:net.W를 인수로 받아 손실함수를 계산하는 새로운 함수
-dW:기울기
-w를 h만큼 늘리면 손실함수의 값은 '결과값*h'만큼 감소함

4.5

학습 알고리즘 구현

1.미니배치(무작위)
2.기울기산출(손실함수의 값을 가장 작게하는 방향 제시)
3.매개변수갱신(기울기방향으로 조금 갱신)
4.반복

입력층 뉴런수 은닉층뉴런수 출력층 뉴런수

```
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
```

첫번째층가중치

첫번째층 편향

2번째층

예측

손실

2층 신경망 클래스

```
    # 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
```

가중치 매개변수 기울기

첫번째층 가중치의 기울기

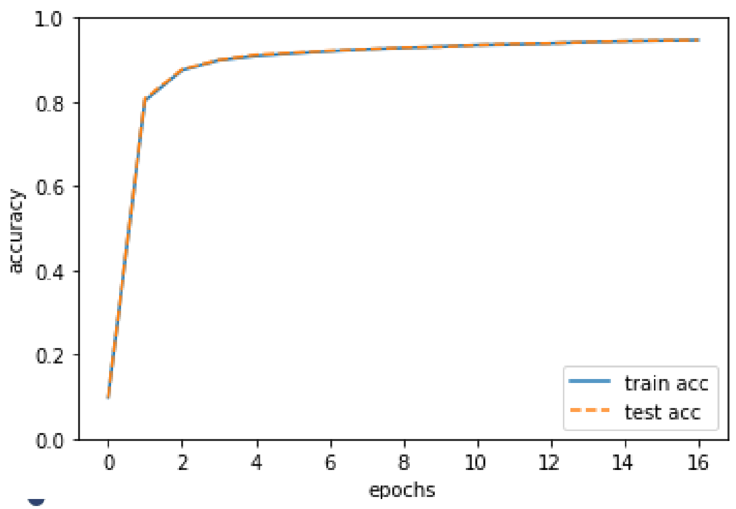
편향의 기울기

2번째층

4.5

미니배치 학습 구현

```
# 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))
```

100개의 데이터를 선택하여 경사하강법 수행하며 매개변수를 갱신, 갱신횟수는 10000번

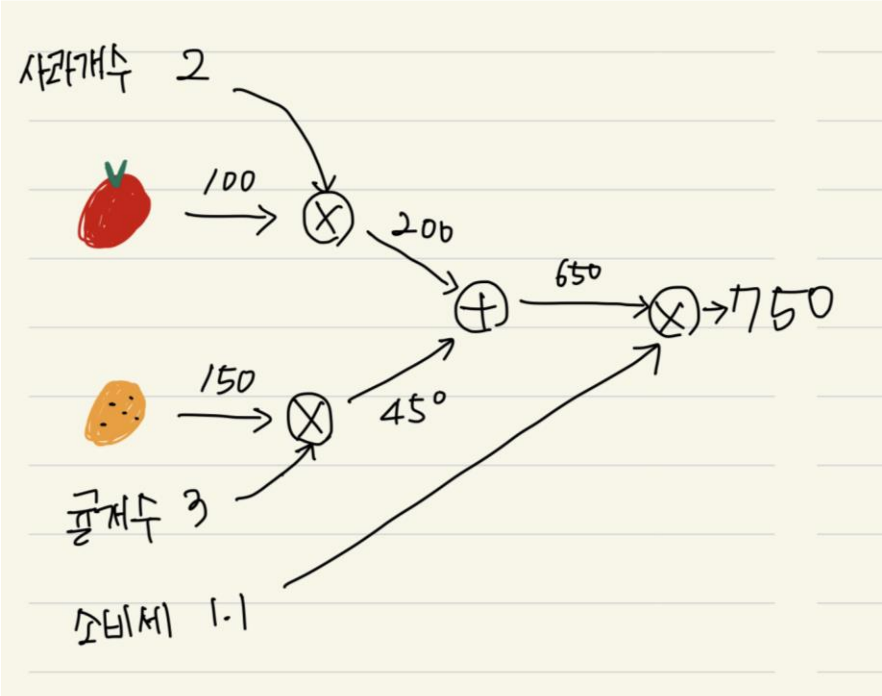
에폭: 훈련데이터를 모두 소진했을 때의 횟수, 훈련데이터가 10000개라면 100회가 1에폭(100*100=10000)

-에폭이 진행될수록 정확도가 좋아짐. 이는 오버피팅이 일어나지 않는다는 것.

머신러닝 5.오차역전파법

오차전역파법

4장에서는 신경망의 가중치 매개변수의 기울기를 수치미분을 이용함.
단순하고 구현하기 쉽지만 계산시간이 오래걸림
-오차전역파법은 이 기울기를 더욱 효율적으로 계산



사과를 2개 귤을 3개 샀을 때, 사과는 개당 100원, 귤은 개당 150원 소비세는 10%

-계산이 왼쪽에서 오른쪽으로 진행하는 순전파
-오른쪽에서 왼쪽은 역전파
-원들이 노드

각 노드는 자신과 관련한 계산만 신경쓴다.
국소적 계산-> 국소적 계산이 모여 전체의 복잡한 계산

역전파:사과 가격에 대한 지불금액의 미분값을 국소계산을 통해 구함

$$z = (x+y)^2$$

$$z = t^2$$

$$t = x + y$$

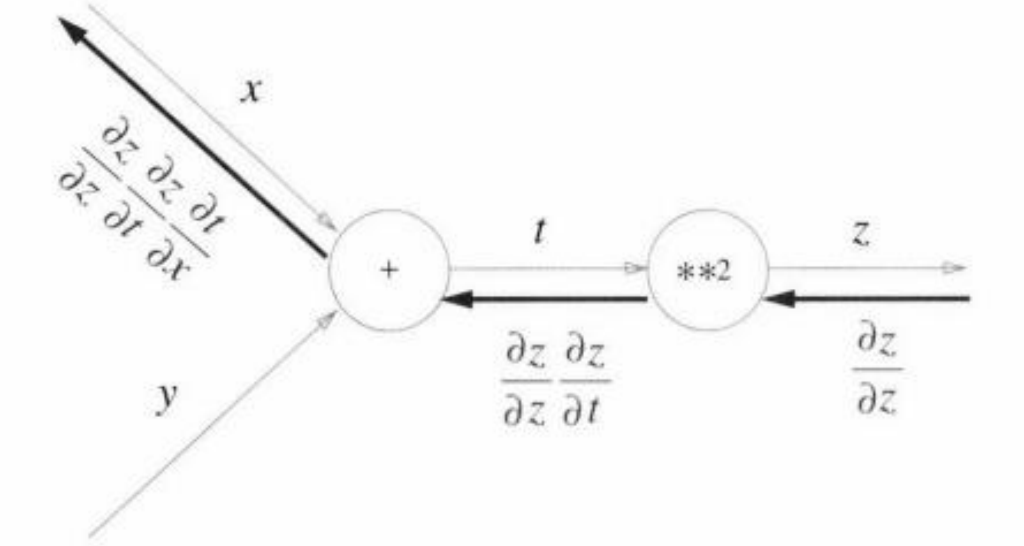
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\cancel{\partial t}} \frac{\cancel{\partial t}}{\partial x}$$

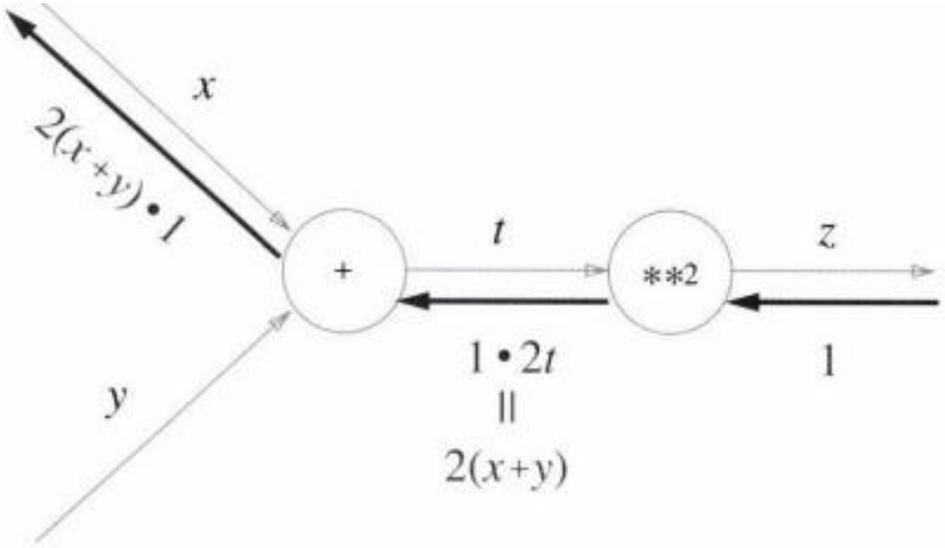
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x+y)$$

연쇄법칙
국소적 미분을 전달하는 원리

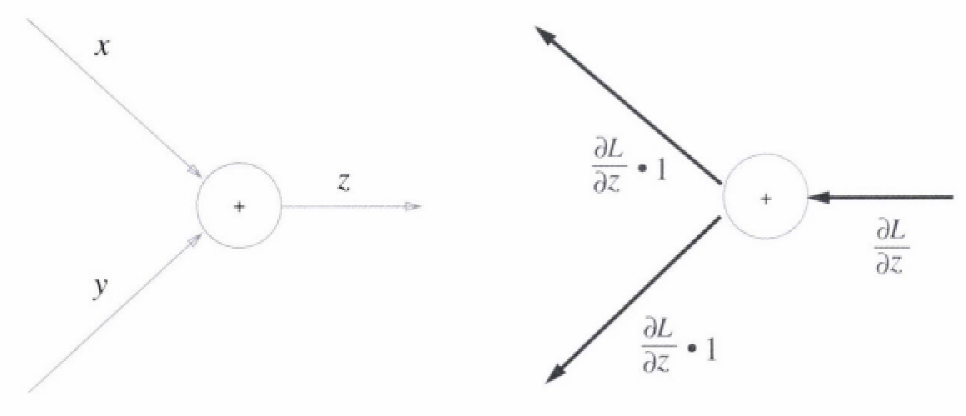
합성함수: 여러 함수로 구성된 함수

연쇄법칙: 합성함수의 미분은 합성함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.





5.3

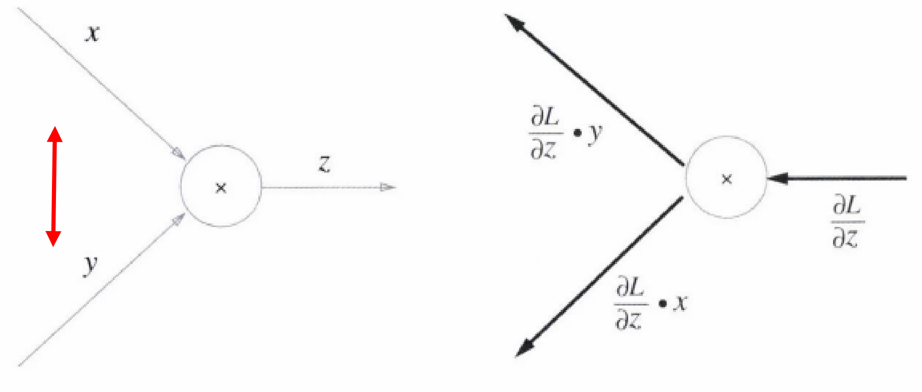


덧셈노드의 역전파

Z= x+y

입력된 값을 그대로 다음 노드로 보냄

L: 최종적으로 L이라는 값을 출력하는 큰 계산그래프를 가정하기 때문



곱셈노드의 역전파

Z= xy

$$\frac{\partial z}{\partial x} = y$$

$$\frac{\partial z}{\partial y} = x$$

5.4

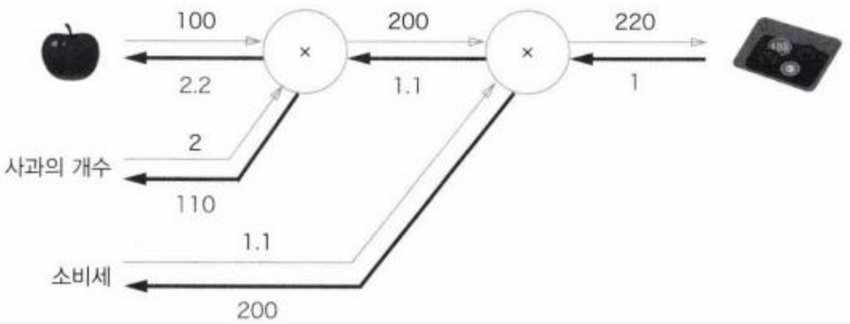
```python
class MulLayer:
    def __init__(self):
        self.x = None
        self.y = None

    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
```

단순계층

MulLayer: 곱셈노드, Addlayer: 덧셈노드
Forward(): 순전파, backward():역전파

곱셈계층

X와 y 초기화
Dout미분(순전파 때의 값을 서로 바꿔 곱한 후 하류에 흘림)




```python
from deeplearning5.layer_naive import *


apple = 100
apple_num = 2
tax = 1.1

mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num)
price = mul_tax_layer.forward(apple_price, tax)

# backward
dprice = 1
dapple_price, dtax = mul_tax_layer.backward(dprice)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)

print("price:", int(price))
print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dTax:", dtax)
```

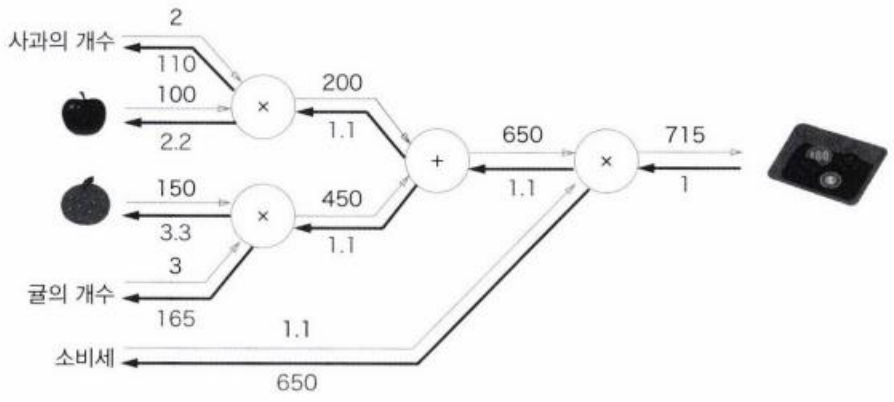
```
price: 220
dApple: 2.2
dApple_num: 110
dTax: 200
```

5.4

덧셈계층
초기화는 아무일도 하지 않으며
backward()는 미분을 그대로 하류에 흘림

```python
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 5-17 사과 2개와 귤 3개 구입




```python
from deeplearning5.layer_naive import *

apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1

# layer
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()

# forward
apple_price = mul_apple_layer.forward(apple, apple_num)  # (1)
orange_price = mul_orange_layer.forward(orange, orange_num)  # (2)
all_price = add_apple_orange_layer.forward(apple_price, orange_price)  # (3)
price = mul_tax_layer.forward(all_price, tax)  # (4)

# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice)  # (4)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price)  # (3)
dorange, dorange_num = mul_orange_layer.backward(dorange_price)  # (2)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)  # (1)

print("price:", int(price))
print("dApple:", dapple)
print("dApple_num:", int(dapple_num))
print("dOrange:", dorange)
print("dOrange_num:", int(dorange_num))
print("dTax:", dtax)
```

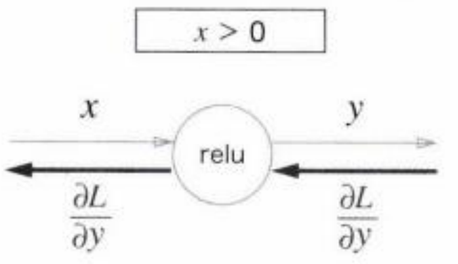
```
price: 715
dApple: 2.2
dApple_num: 110
dOrange: 3.3000000000000003
dOrange_num: 165
dTax: 650
```

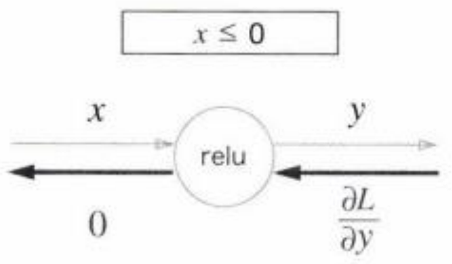
활성화 함수계층 구현하기

RelU계층

$$y=\begin{cases} x & (x>0) \\ 0 & (x\le 0)\end{cases}$$

$$\frac{\partial y}{\partial x}=\begin{cases} 1 & (x>0) \\ 0 & (x\le 0)\end{cases}$$

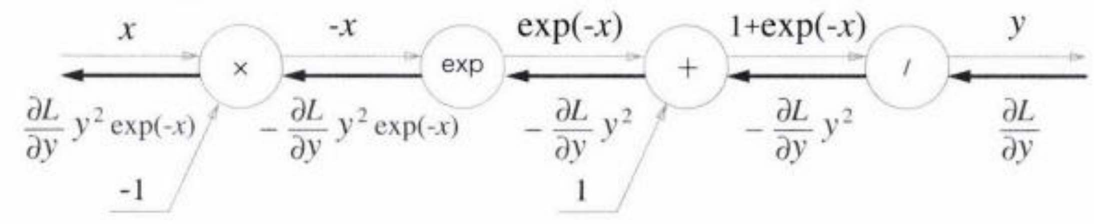


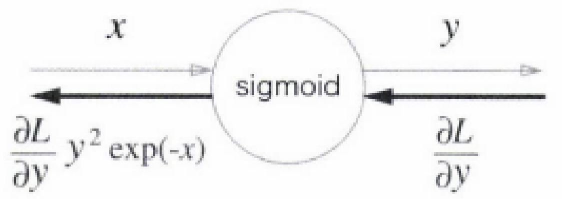


Sigmoid계층

-Exp노드는 y= exp(x)
-'/'노드는 y=1/x

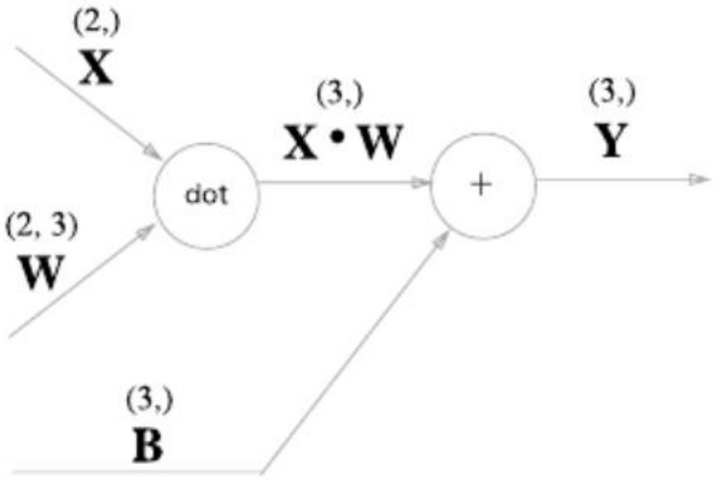
$$y=\frac{1}{1+\exp(-x)}$$





Affine 계층

Affine 변환
-기하학에서 신경망의 순전파 때 수행하는 행렬의 곱
-어파인변환을 수행하는 처리를 affine 계층이라는 이름으로 구현



$$\frac{\partial L}{\partial \mathbf{X}} = \frac{\partial L}{\partial \mathbf{Y}} \cdot \mathbf{W}^{\mathrm{T}}$$

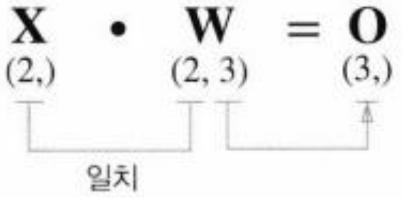
$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathbf{Y}}$$

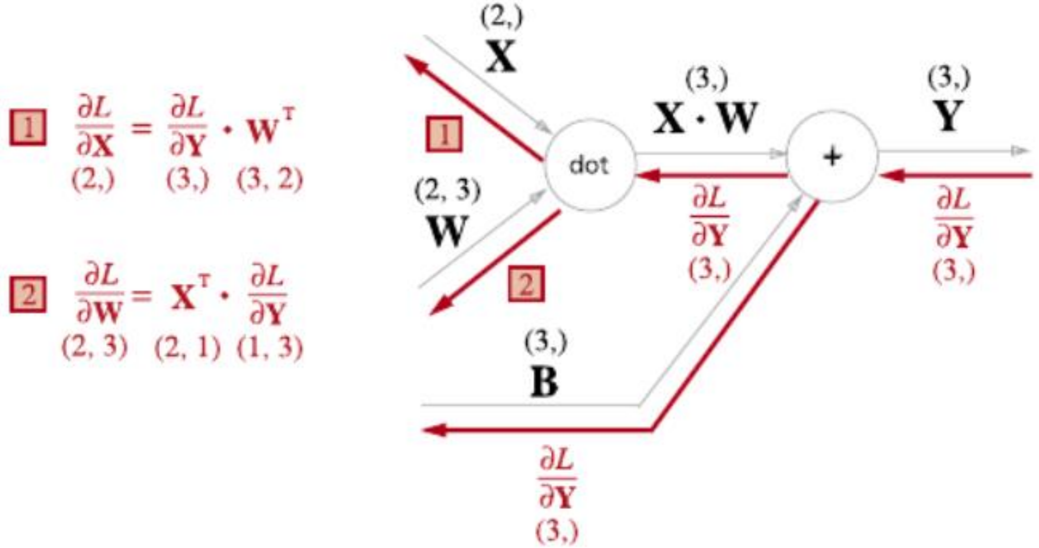
$$\mathbf{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$$

전치행렬 $\mathbf{W}^{\mathrm{T}} = \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{pmatrix}$

-변수는 행렬
-X= 입력
-W= 가중치
-B=편향
(각각의 다차원배열)
-뉴런의 가중치 합 Y=np.dot(X,W)+B
-행렬의 곱의 역전파는 행렬의 대응하는 차원의 원소 수가 일치하도록 곱을 **조립**하여 구함

그림 5-23 행렬의 곱에서는 대응하는 차원의 원소 수를 일치시킨다.



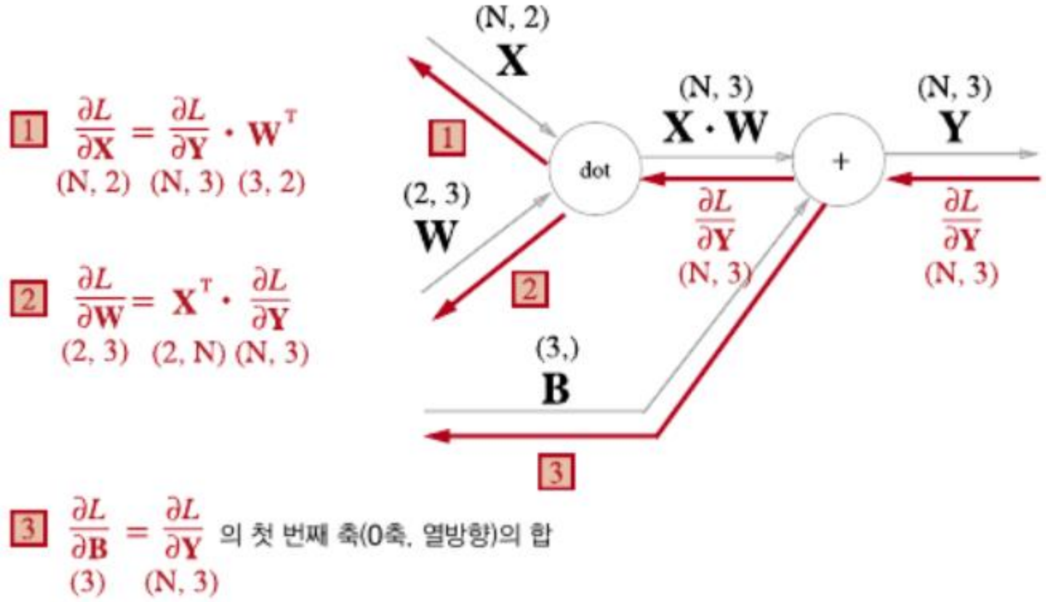


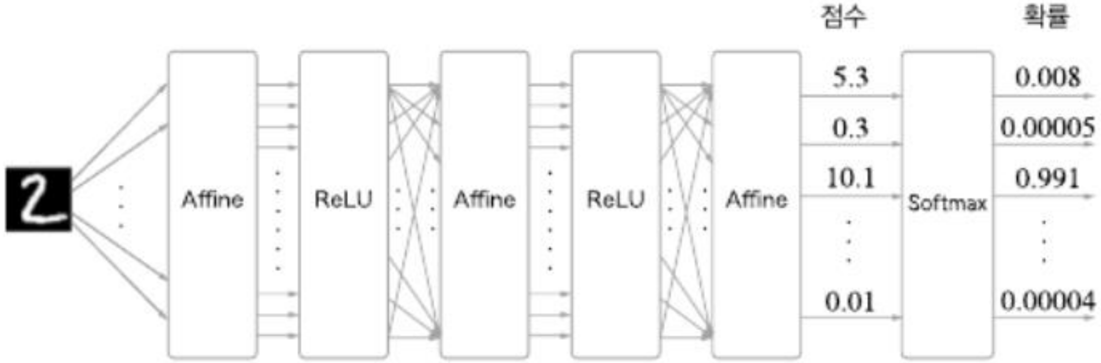
배치용affine 계층

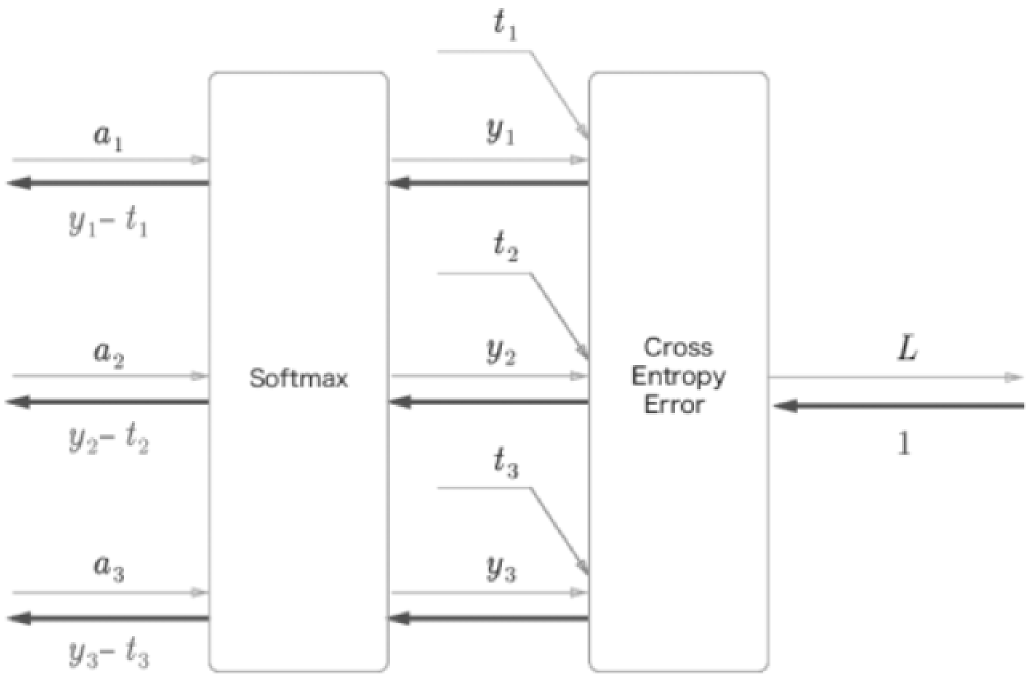
X하나만을 고려하는 것이 아닌 데이터N개를 묶어 순전파하는 경우
-즉 입력 X의 형상이 (N,2)가 된 것!

주의할점: 편향을 더할 때 데이터가 2개인 경우 편향은 두 데이터 각각에 더해진다.

역전파 때는 각데이터의 역전파값이 편향의 원소에 모여야함(두 데이터에 대한 미분을 데이터마다 더해서 구함)







softmax-with-loss 계층

-출력층에서 사용함
-소프트맥스 함수는 입력값을 정규화 해 출력
-마지막 softmax 계층에 의해 입력이 정규화 되며 이 때 손실함수인 교차엔트로피 오차도 포함한 것이 바로 softmax-with-loss 계층
-매우 복잡하므로 결과만을 제시하고 있음

-Softmax계층은 입력(a1,a2,a3)를 정규화해서 (y1,y2,y3) 출력
-Cross entrophy error 계층은 이 출력레이블과 정답레이블인 (t1,t2,t3)를 받고 이것들로부터 손실 L을 출력
-역전파는 softmax계층의 출력값과 정답레이블의 차분

즉! 신경망의 역전파에서는 오차가 앞 계층에 전해지는 것.

따라서 오차가 클수록 제대로 정답을 인식하지 못하였으므로 크게 학습하는 것이고 오차가 작을수록 정답에 가까우므로 학습하는 정도도 작은 것

```python
# 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()
```

오차역전파법 구현

4장과의 2층 신경망과 다른점: 계층을 사용한다는 것!

```python
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
```

OrderedDict은 순서가 있는 딕셔너리
-순전파: 추가한 순서대로 메서드를 호출
-역전파: 반대순서로 호출

Affine 계층과 RelU계층이 순전파와 역전파를 처리해주고 있음

학습 구현

기울기를 구하는 두가지 방법
1.**수치미분**
2.**오차역전파법**

수치미분의 이점: 구현이 쉬워 버그가 적음

따라서 오차역전법을 제대로 구현했는지 검증하기 위해 두 결과를 비교함(기울기확인)
-기울기의 차이가 작으면 오차역전파법을 실수없이 구현한 것!

Network.gradient: 오차역전파법

-기울기 계산식에 수치미분방식을 넣을지 오차역전파법 방식을 넣을지에 따라 다른 것!

```python
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 부모 디렉터리의 파일을 가져올 수 있도록 설정
import numpy as np
from deeplearning_dataset.mnist import load_mnist
from deeplearning5.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]

grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)

# 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))
```

```
W1:4.3889505264684874e-10
b1:2.480753748451475e-09
W2:5.465435361172855e-09
b2:1.3959483319281318e-07
```