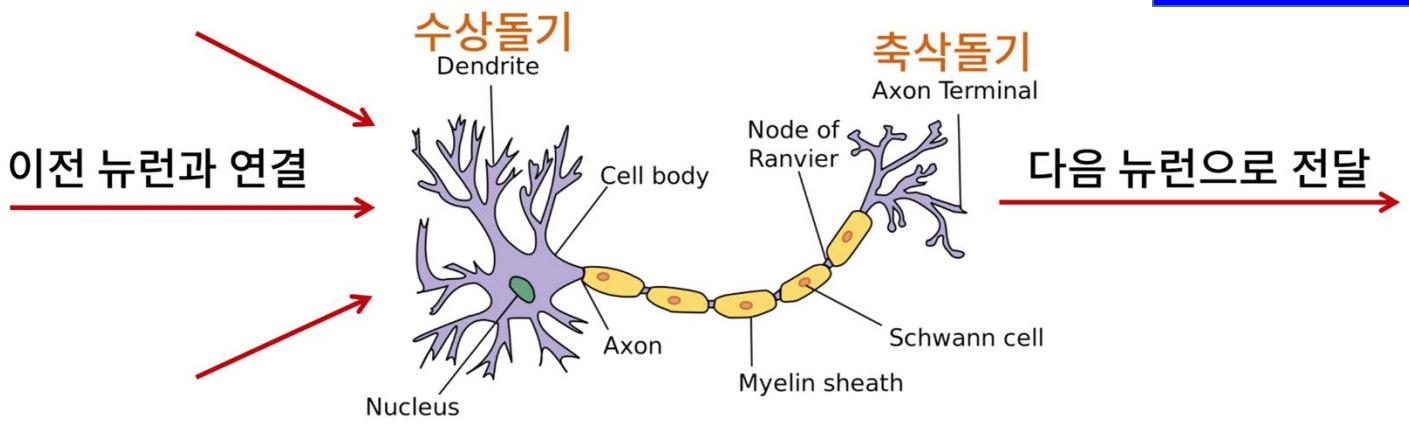
인공지능 이론

신경망의 이해

뉴런(Neuron)과 퍼셉트론(Perceptron)

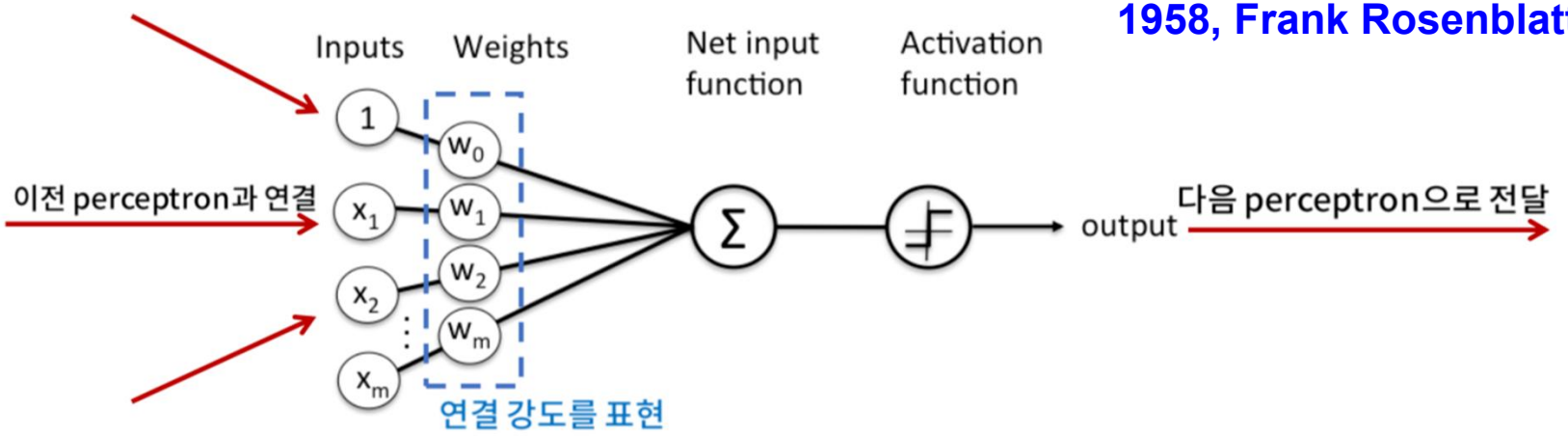
생물학적 뉴런



인공신경망 퍼셉트론

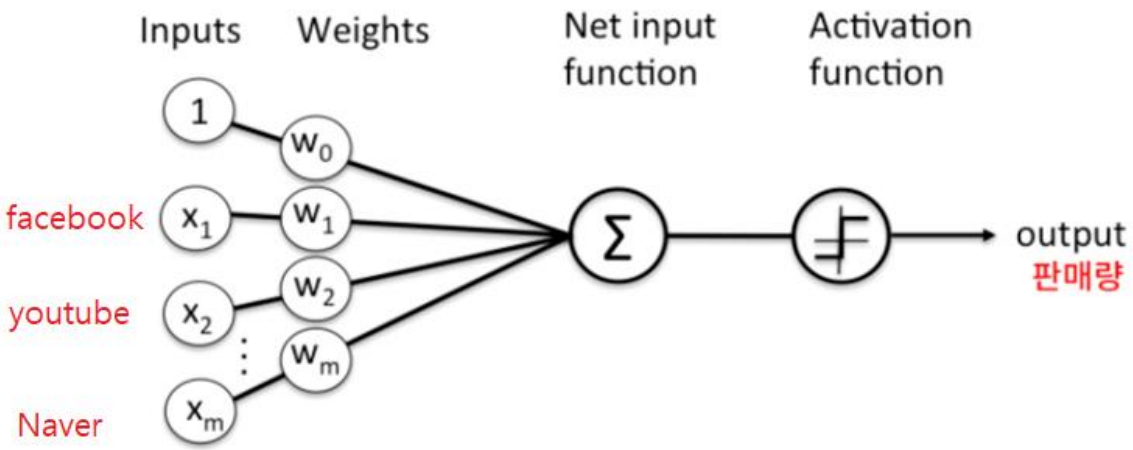
생물학적 뉴런을 모방하여 만든 인공신경망의 기본 단위

1958, Frank Rosenblatt



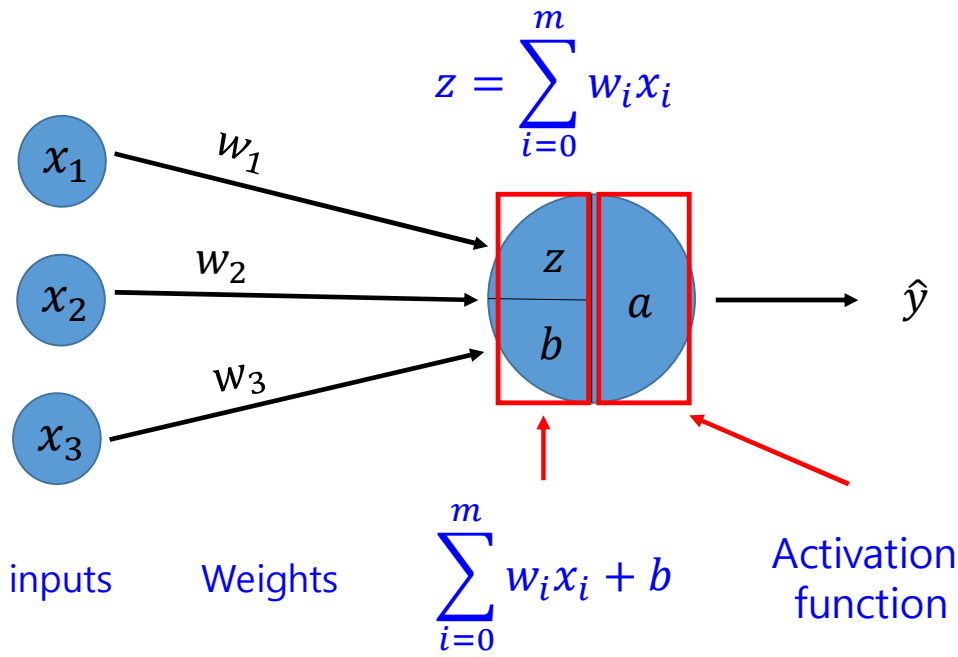
퍼셉트론의 간단한 예

전자제품을 판매하는 회사에서 광고 매체에 따른 판매량을 예측

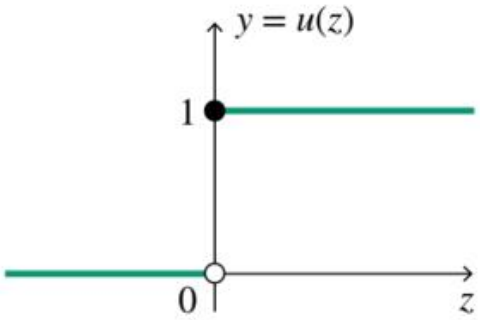


facebook	Youtube	Naver	판매량
120	68	50	210
40	22	30	111
17	46	70	73
81	40	57	181
57	33	23	122

퍼셉트론의 구성 요소


$z = \sum_{i=0}^{m} w_i x_i$
x_1
x_2
x_3
w_1
w_2
w_3
z
b
a
$\hat{y}$
inputs
Weights
$\sum_{i=0}^{m} w_i x_i + b$
Activation function

활성화 함수의 원형 – 단위 계단 함수

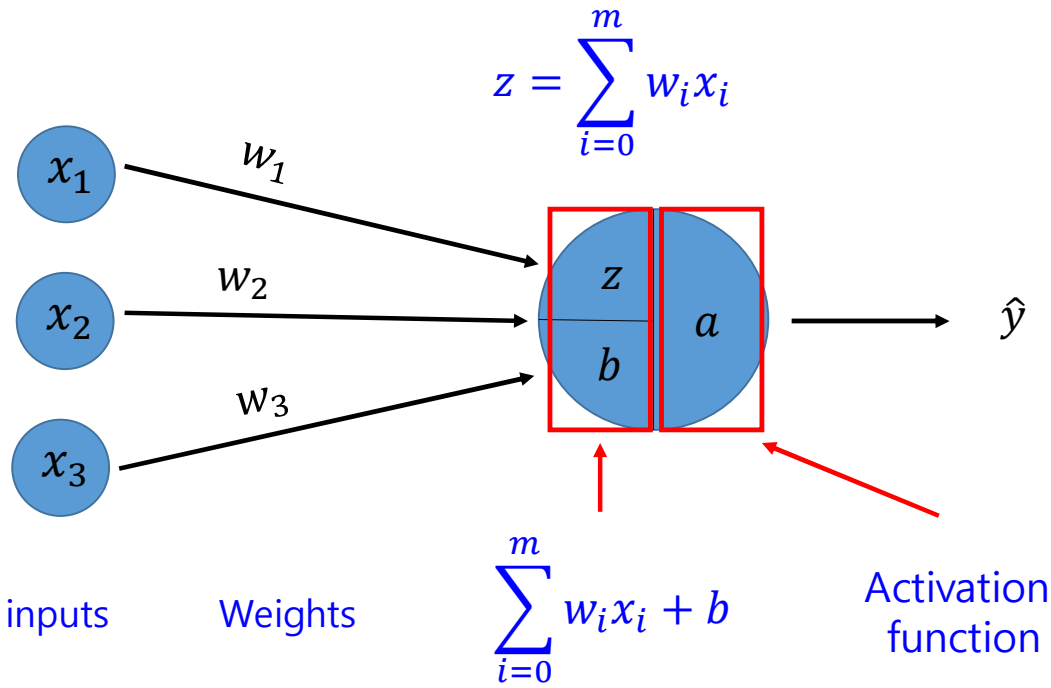


단위 계단 함수 $y = u(z)$

$$u(z) = \begin{cases} 0 & (z < 0) \\ 1 & (z \geqq 0) \end{cases}$$

y	$w_1x_1+w_2x_2+w_3x_3$	$z = w_1x_1+w_2x_2+w_3x_3-\theta$	$u(z)$
0(반응하지 않음)	θ보다 작음	$z < 0$	0
1(반응함)	θ보다 큼	$z \geqq 0$	1

활성화 함수의 원형 – 단위 계단 함수

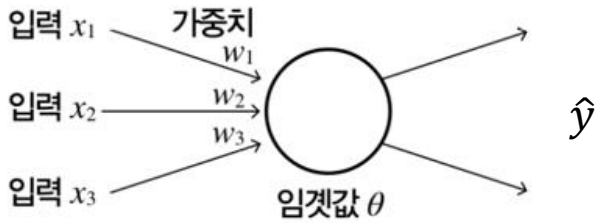


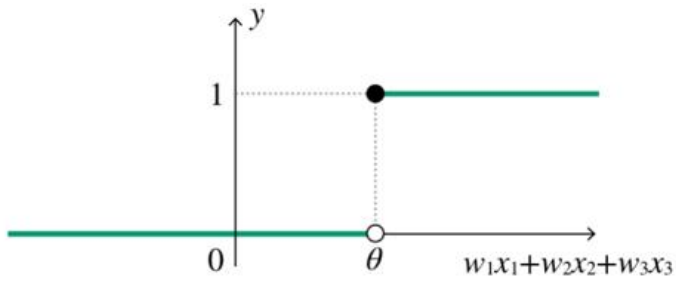
inputs

Weights

Activation function

비선형적인 특성을 만들어준다.





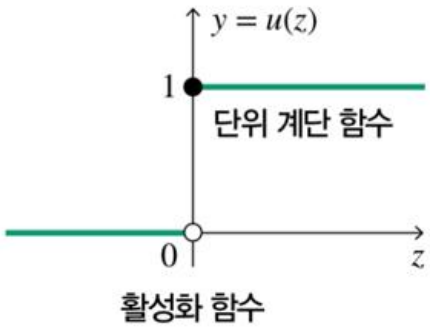
출력 신호 없음(y=0) : $w_1x_1 + w_2x_2 + w_3x_3 < \theta$

출력 신호 있음(y=1) : $w_1x_1 + w_2x_2 + w_3x_3 \geq \theta$

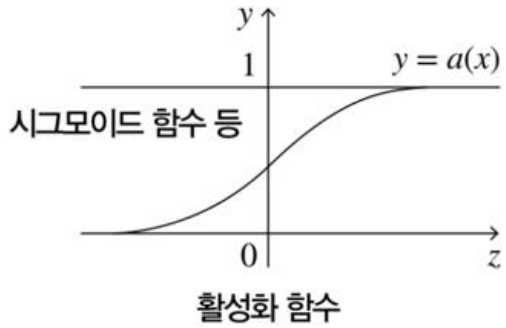
단위 계단 함수 VS 시그모이드 함수

활성화 함수의 가장 기본적인 초기 모델은 시그모이드 함수이다.

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad (e = 2.718281\cdots)$$



활성화 함수

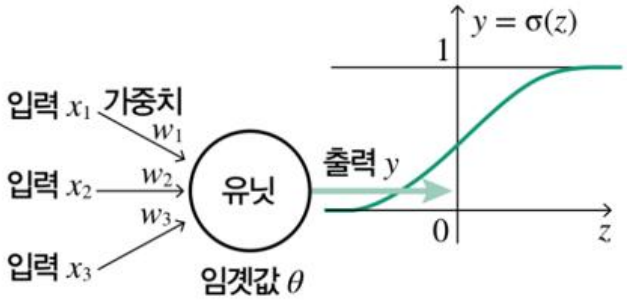


활성화 함수

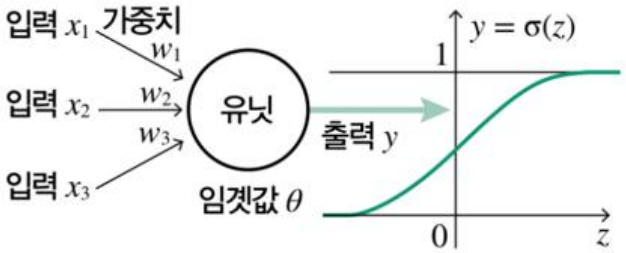
단위 계단 함수는 불연속적이다 → 미분 불가능

활성화함수는 비선형 함수를 사용한다.

단위 계단 함수 VS 시그모이드 함수



유닛의 흥분도가 낮음



유닛의 흥분도가 높음

신경망에서는 활성화 함수로 비선형 함수를 사용한다. 선형 함수를 사용하지 않는 이유는 선형 함수를 사용하면 신경망의 층을 깊게 하는 의미가 없어지기 때문이다.

선형 함수의 문제는 층을 아무리 깊게 해도 '은닉층이 없는 네트워크'로도 똑같은 기능을 할 수 있다는데 있다.

단위 계단 함수 VS 시그모이드 함수

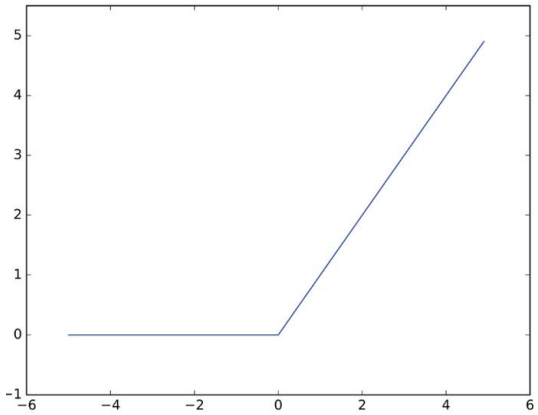
$$f(kx) = kf(x) \qquad f(x1 + x2) = f(x1) + f(x2)$$

선형 함수인 $h(x) = cx$ 를 활성화 함수로 사용한 3층 네트워크를 생각해보자. 이를 식으로 나타내면 $y(x) = h(h(h(x)))$가 된다. 이 계산은 $y(x) = c * c * c * x$ 처럼 곱셈을 세 번 수행하지만, 실은 $y(x) = ax$와 똑같은 식이 된다. 즉, 은닉층이 없는 네트워크로 표현 할 수 있다.

활성화 함수 : ReLU

시그모이드 함수는 신경망 분야에서 오래전부터 사용해 왔으나, 2010년 AlexNet이후에는 ReLU(Rectified Linear Unit)함수를 주로 이용한다.

ReLU는 입력이 0을 넘으면 그 입력을 그대로 출력하고, 0이하이면 0을 출력하는 함수이다.



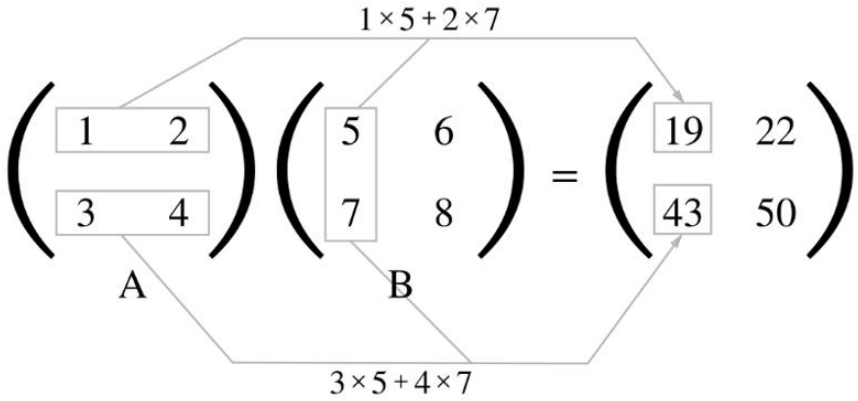
$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

ReLU는 0보다 작을 때는 '0', 0보다 클 때는 x값을 갖는다. 외형적인 것만 놓고 보면 '선형함수가 아닌가?' 할 수 있지만 실제 선형 함수의 정의로 확인해보면 선형 함수가 아니다.

$f(kx) = kf(x)$ $\qquad$ $f(x1 + x2) \neq f(x1) + f(x2)$

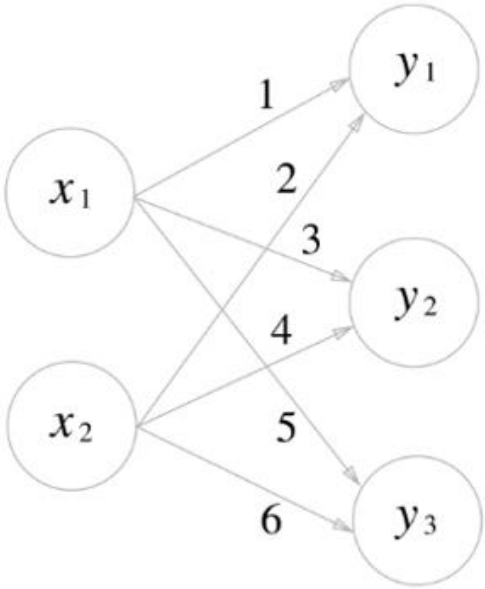
동차성은 성립한다. $\qquad$ **가산성은 성립하지 않는다.**

행렬의 곱




```
>>> A = np.array([[1,2], [3,4]])
>>> A.shape
(2, 2)
>>> B = np.array([[5,6], [7,8]])
>>> B.shape
(2, 2)
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

신경망에서의 행렬 곱




```
>>> X = np.array([1, 2])
>>> X.shape
(2,)
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> print(W)
[[1 3 5]
 [2 4 6]]
>>> W.shape
(2, 3)
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5  11  17]
```

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

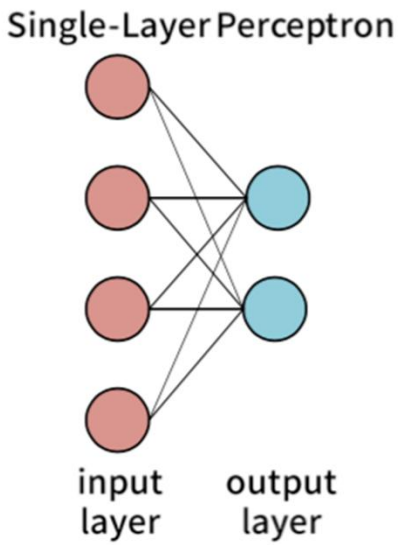
X	W	=	Y
2	2×3		3

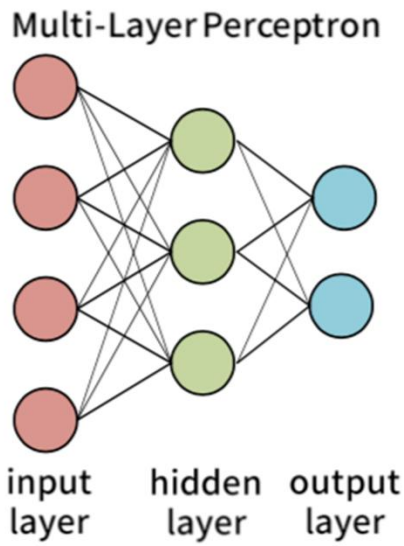
일치

인공 신경망(Artificial Neural Network : ANN)

Artificial Neural Network
Perceptron을 여러 개 연결한 것

Artificial Neural Network의 학습
우리가 원하는 결과를 위해 weight값들을 찾아내는 과정

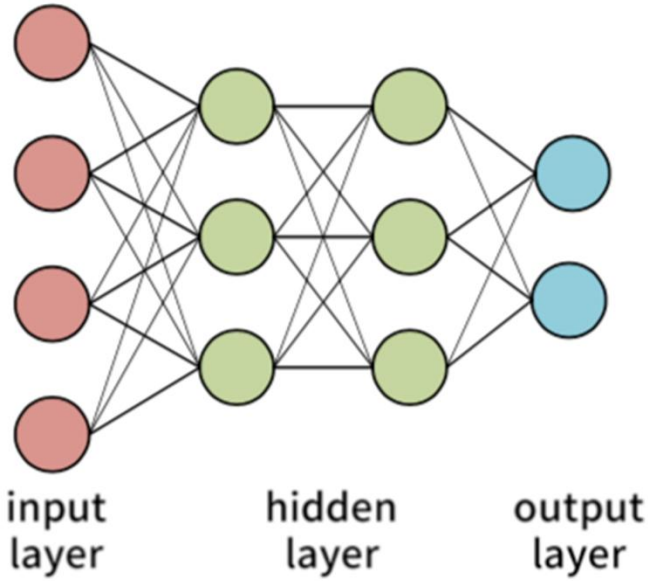




딥러닝 (Deep Neural Network)

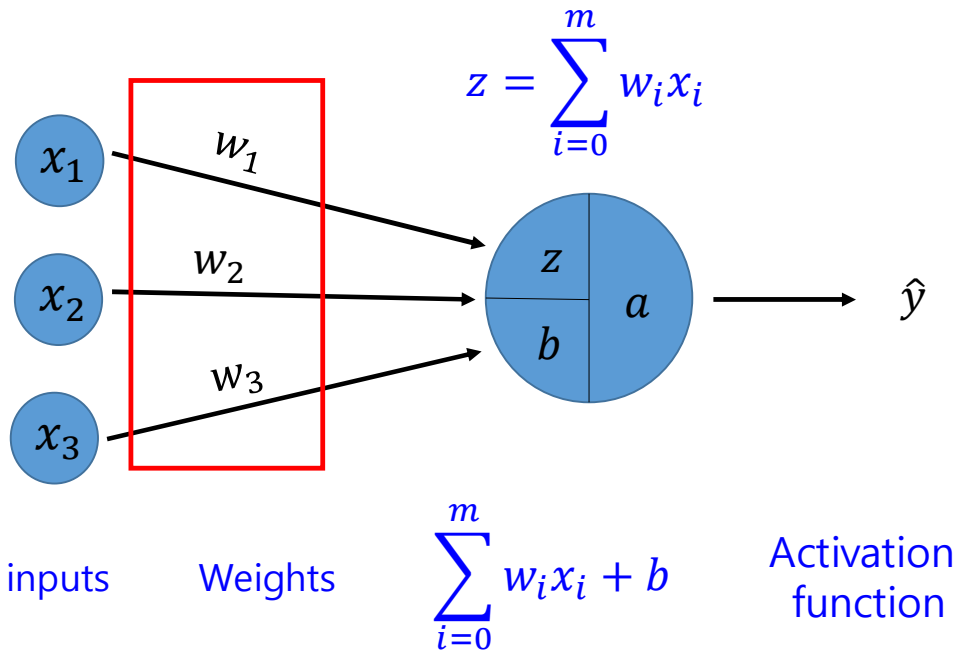
Deep Neural Network을 이용한 Machine Learning 방법

Deep Neural Network(DNN) : Hidden layer 여러 개의 은닉층들로 이루어진 인공
신경망이다. 심층 신경망은 일반적인 인공신경망과 마찬가지로 복잡한 비선형 관계들을 모델링 할 수 있다.



Weight 값의 초기화

Weight값은 어떤 값으로 초기화 하는 것이 좋을까?



Weight 값의 초기화

Weight값은 어떤 값으로 초기화 하는 것이 좋을까?

Zero Initalize
가장 쉽게 생각할 수 있는 것은 '모든 파라메터를 0으로 놓고 학습을 시작하면 되지 않을까'라고 생각 할 수 있다. 그러나 파라메터의 값이 모두 같다면 역전파(back propergation)을 통해서 weight값이 모두 같은 값으로 업데이트 될 수 있다. 그렇게 되면 제대로 학습이 이루어지지 않을 가능성이 있어서 좋은 초기화는 아니다.

Random Initialization
서로 다른 값을 부여하기 위해서 쉽게 생각할 수 있는 방법은 확률 분포에 따른 난수값(Random value)으로 가중치를 초기화하는 것이다.

그이외에 Xavier 초기화나 He초기화가 있으나, 우리는 일단 radom값으로 초기화 하자.

https://reniew.github.io/13//

Neural Network가 실제 값과 얼마나 비슷한지에 대한 척도

(뉴럴넷 예측값 – 정답값)^2

2가지 광고료 조합에 대하여 Neural Network은

예측한 판매량 : (100, 80)
실　제 판매량 : (105, 78)

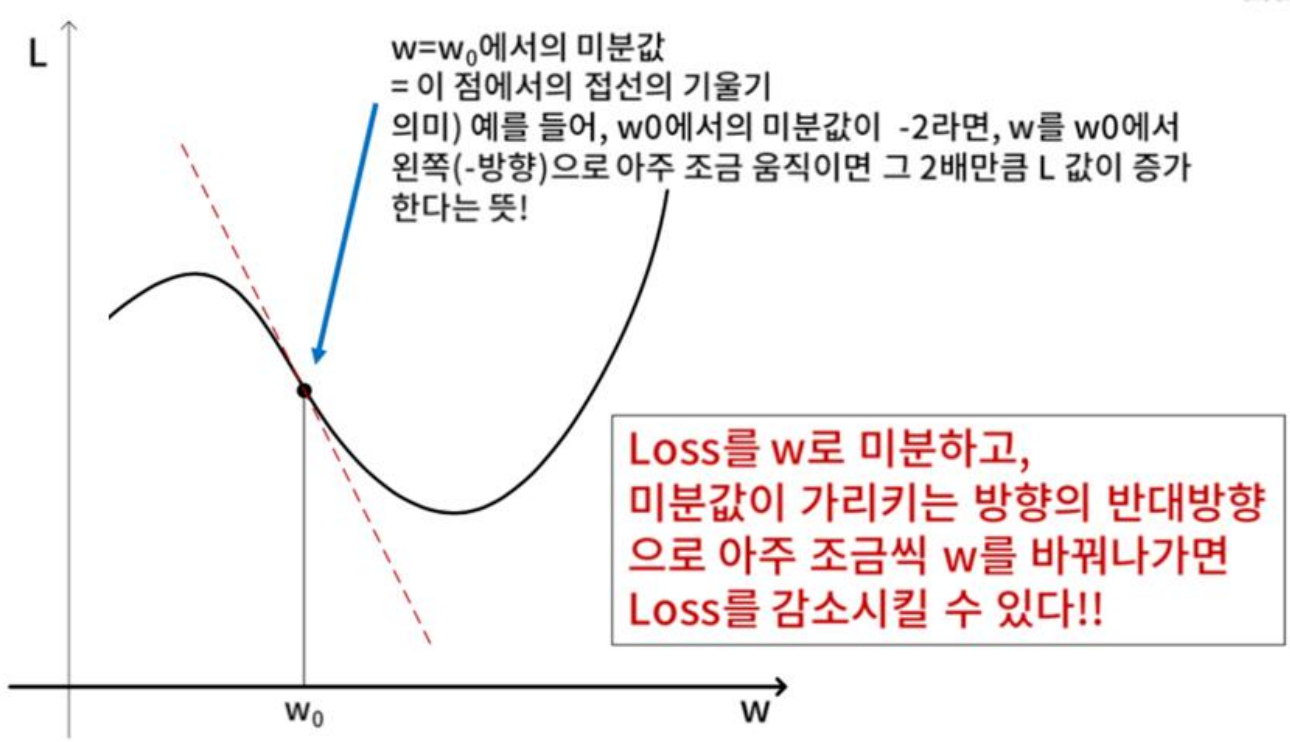
→ $L = (105-100)^2 + (78-80)^2 = 29$

예측 값의 정확도를 높이기 위해 필요한 것은
Cost Function의 값이 줄어들도록 weight값들을 조금씩 바꾸는 것

Weight를 어떻게 바꾸어야
Cost Function값이 줄어들까?

→ **미분**

Cost Function에서 최소값을 구하려면



Gradient Descent (경사 하강법)

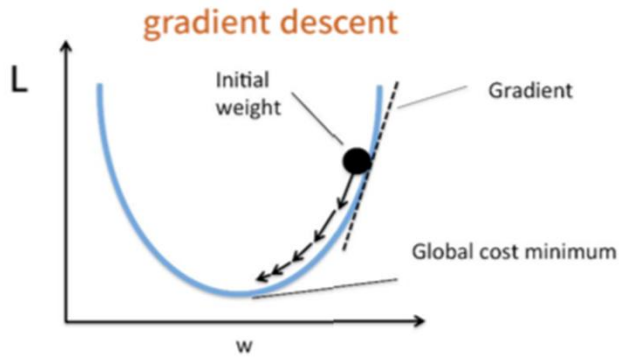
1차 근사값 발견용 최적화 알고리즘이다. 기본 아이디어는 함수의 기울기(경사)를 구하여 기울기가 낮은 쪽으로 계속 이동시켜서 극값에 이를 때까지 반복한다.

$$w_{new} = w_{old} - \eta \nabla_w L$$

Weight update $= w_{new} - w_{old}$

$= -\eta \nabla_w L$

- $-$: Loss를 감소시키는 방향 (Descent)
- η : 아주 조금씩 이동 (Learning Rate)
- $\nabla_w L$: 미분값 (Gradient)



Gradient Descent (경사 하강법)

1. **모든 Training Data에 대해**
   Neural Network의 출력과 실제 Lable을 비교하여 각각의 Loss(Cost)값을 계산한다.

2. **이 Loss를 Weight로 미분한 다음**
   그 미분값(Gradient)이 가리키는 방향의 반대 방향으로 weight값을 아주 조금씩 바꿔 나간다.

문제점 : 데이터의 양이 많은 경우 학습 속도가 느리다!

Batch/Mini-batch/Stochastic Gradient Descent

모든 training data에 대하여 Loss를 계산하면 시간이 너무 오래 걸림. 개선 방법은…

Batch Gradient Descent

전체 데이터를 사용하여 , Loss를 계산하고, Gradient Descent(경사하강법)를 행한다.

Mini-batch Gradient Descent

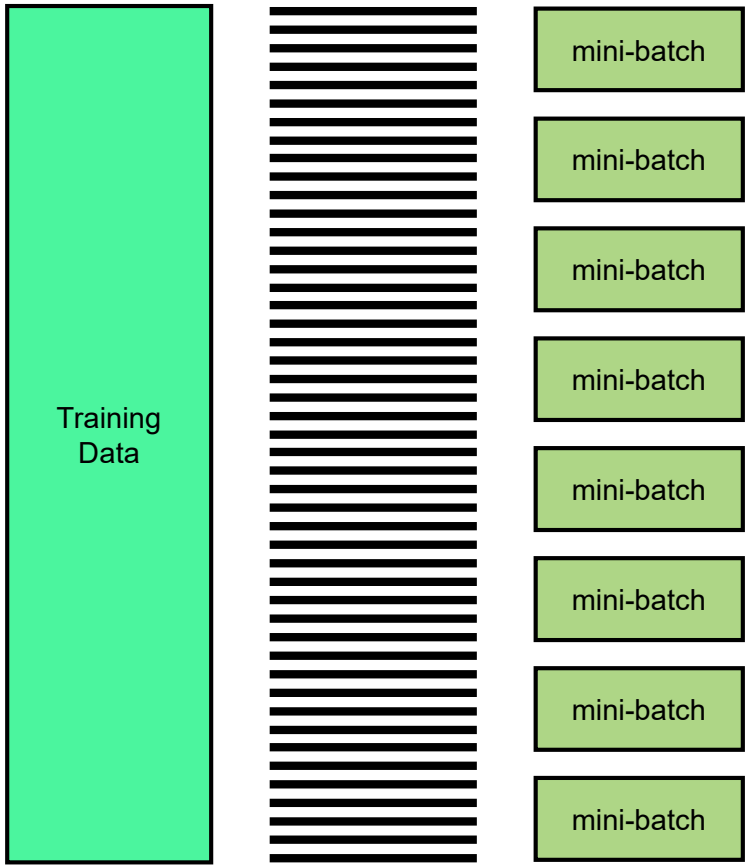
Batch/Stochastic의 중간 형태로 data를 n개 뽑고 그 n개의 data에 대한 Loss를 계산하여

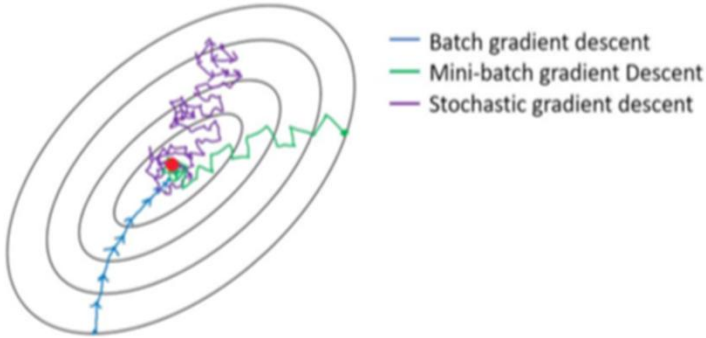
다 더한 뒤 이를 이용하여 Gradient Descent를 행한다.

Stochastic Gradient Descent (랜덤하기 확률적으로 Data를 취한다.)

Data를 1개만 선택하고, 1개의 data에 대한 Loss를 이용하여 Gradient Descent를 행한다.

Batch/Stochastic/Mini-batch Gradient Descent





Backpropagation(오차역전파법)

오차역전파법을 이해하는 방법은 2가지가 있다.

1. 수식을 통해 이해하는 방법
2. 계산그래프를 통하는 방법

수식을 사용한 설명은 정확하고 간결하다. 그러나 수많은 수식에 당황하는 일도 있다.

반면, 계산 그래프를 이용하면 시각적으로 이해하는데 도움이 된다.

계산 그래프는 계산 과정을 그래프로 나타낸 것이다.

그래프는 우리가 자주 사용하는 자료구조로 복수의 노드와 에지로 구성된다.

(노드 사이의 직선을 에지라고 볼 수 있다.)

수학으로 이해하는 오차역전파법

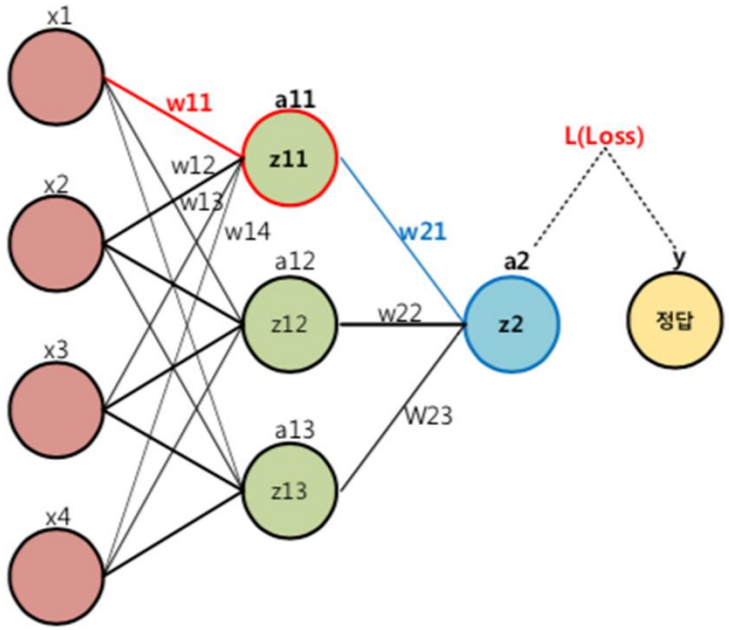
$$z11 = x1 \cdot w11 + x2 \cdot w12 + x3 \cdot w13 + x4 \cdot w14$$

$$a11 = \sigma(z11) = \frac{1}{1 + e^{-z11}}$$

$$z2 = a11 \cdot w21 + a12 \cdot w22 + a13 \cdot w23$$

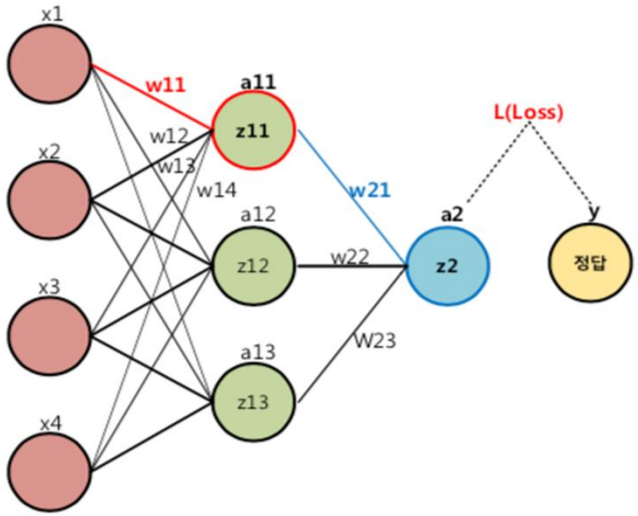
$$a2 = z2$$

$$L = (y - a2)^2$$



수학으로 이해하는 오차역전파법

$$z11 = x1 \cdot w11 + x2 \cdot w12 + x3 \cdot w13 + x4 \cdot w14$$
$$a11 = \sigma(z11) = \frac{1}{1 + e^{-z11}}$$
$$z2 = a11 \cdot w21 + a12 \cdot w22 + a13 \cdot w23$$
$$a2 = z2$$
$$L = (y - a2)^2$$



Loss부터 거꾸로 한 단계씩 미분을 해봅시다

$$\partial L/\partial a2 = 2(y - a2)$$
$$\partial a2/\partial z2 = 1$$
$$\partial z2/\partial a11 = w21$$
$$\partial a11/\partial z11 = \sigma(z11) \cdot \big(1 - \sigma(z11)\big)$$
$$\partial z11/\partial w11 = x1$$

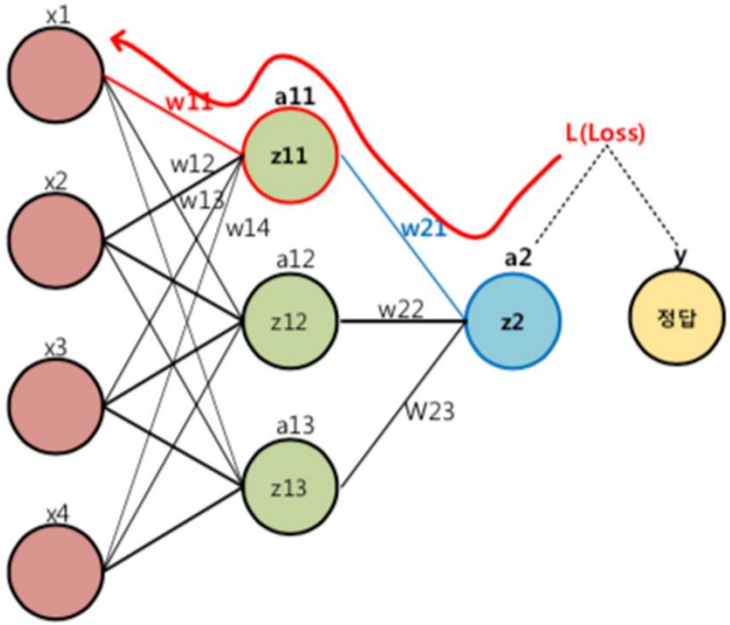
이 미분들을 전부 각각 곱하면(chain rule),

$$\frac{\partial L}{\partial a2} \cdot \frac{\partial a2}{\partial z2} \cdot \frac{\partial z2}{\partial a11} \cdot \frac{\partial a11}{\partial z11} \cdot \frac{\partial z11}{\partial w11} = \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w11}}$$

우리가 구하려고 했던 미분값

수학으로 이해하는 오차역전파법

Loss로부터 거꾸로 한 단계씩 미분 값을 구하고 이 값들을 chain rule에 의하여 곱해가면서 weight에 대한 gradient를 구하는 방법

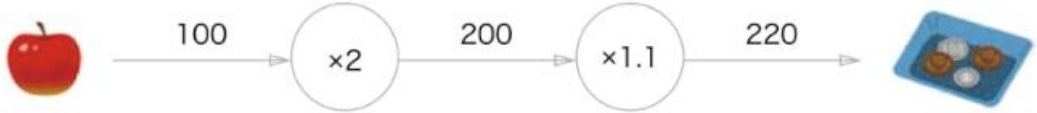


계산 그래프를 통한 방법

문제 1 : 현빈 군은 슈퍼에서 1개에 100원인 사과를 2개 샀습니다. 이때 지불 금액을 구하세요. 단, 소비세가 10% 부과됩니다.

계산 그래프는 계산 과정을 노드와 화살표로 표현합니다. 노드는 원(○)으로 표기하고 원 안에 연산 내용을 적습니다. 또, 계산 결과를 화살표 위에 적어 각 노드의 계산 결과가 왼쪽에서 오른쪽으로 전해지게 합니다. 문제 1을 계산 그래프로 풀면 [그림 5-1]처럼 됩니다.

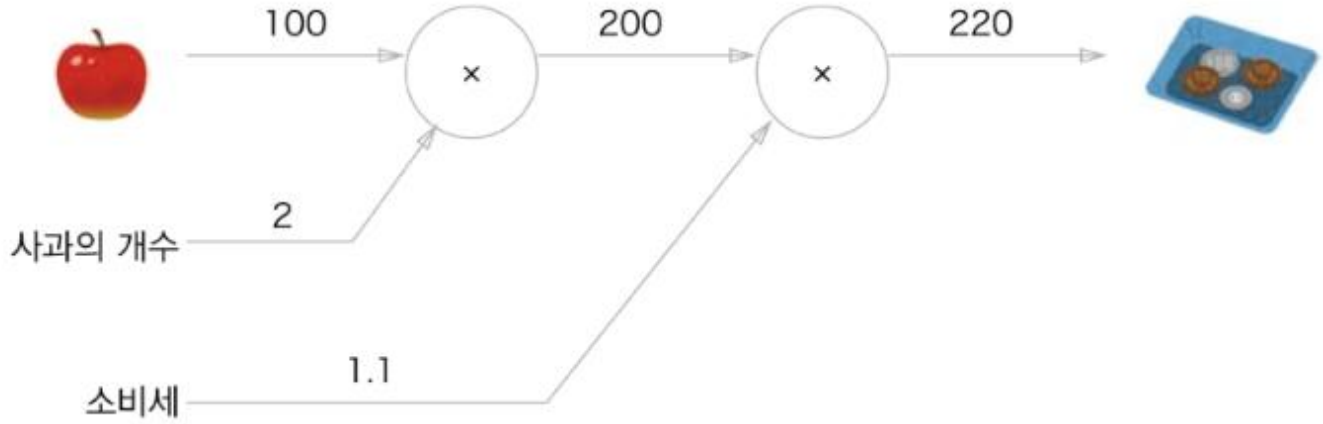
그림 5-1 계산 그래프로 풀어본 문제 1의 답



[그림 5-1]과 같이 처음에 사과의 100원이 '×2' 노드로 흐르고, 200원이 되어 다음 노드로 전달됩니다. 이제 200원이 '×1.1' 노드를 거쳐 220원이 됩니다. 따라서 이 계산 그래프에 따르면 최종 답은 220원이 됩니다.

계산그래프로 풀기

그림 5-2 계산 그래프로 풀어본 문제 1의 답 : '사과의 개수'와 '소비세'를 변수로 취급해 원 밖에 표기



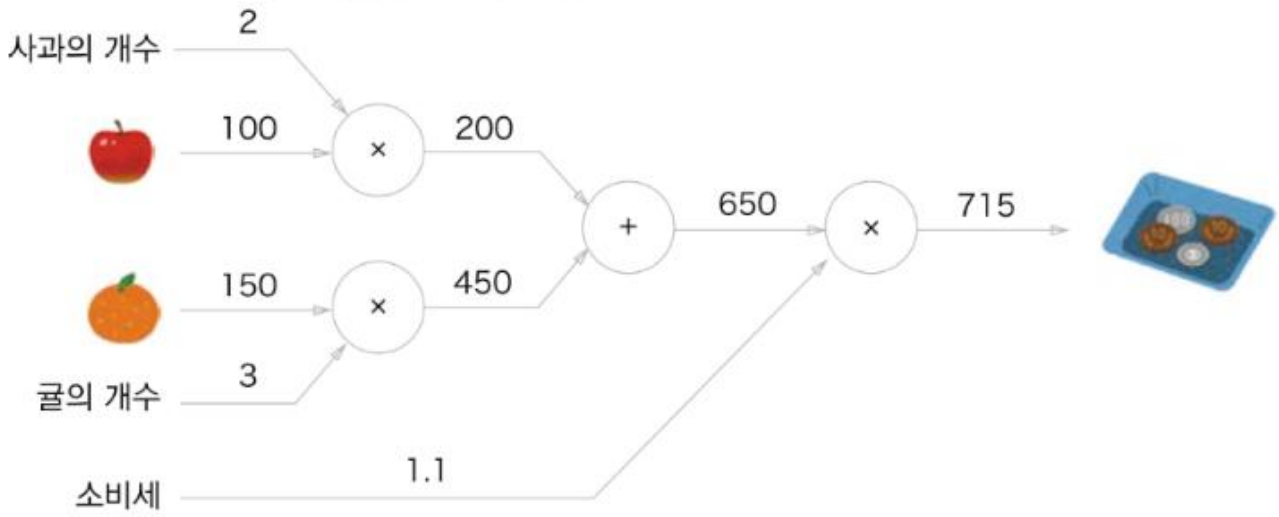
그럼 다음 문제입니다.

계산그래프로 풀기

문제 2 : 현빈 군은 슈퍼에서 사과를 2개, 귤을 3개 샀습니다. 사과는 1개에 100원, 귤은 1개 150원입니다. 소비세가 10%일 때 지불 금액을 구하세요.

문제 2도 문제 1과 같이 계산 그래프로 풀겠습니다. 이때의 계산 그래프는 [그림 5-3]처럼 됩니다.

그림 5-3 계산 그래프로 풀어본 문제 2의 답



계산그래프로 풀기

지금까지 살펴본 것처럼 계산 그래프를 이용한 문제풀이는 다음 흐름으로 진행합니다.

1. 계산 그래프를 구성한다.
2. 그래프에서 계산을 왼쪽에서 오른쪽으로 진행한다.

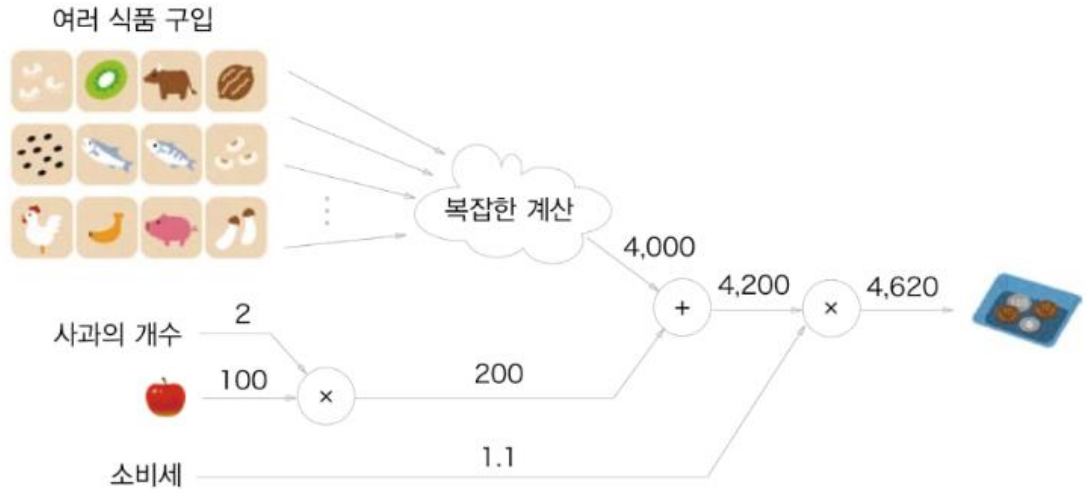
여기서 2번째 '계산을 왼쪽에서 오른쪽으로 진행'하는 단계를 **순전파**forward propagation라고 합니다. 순전파는 계산 그래프의 출발점부터 종착점으로의 전파입니다. 순전파라는 이름이 있다면 반대 방향(그림에서 말하면 오른쪽에서 왼쪽)의 전파도 가능할까요? 네! 그것을 **역전파**backward propagation라고 합니다. 역전파는 이후에 미분을 계산할 때 중요한 역할을 합니다.

국소적 계산

계산 그래프의 특징은 '국소적 계산'을 전파함으로써 최종 결과를 얻는다는 점에 있습니다. 국소적이란 '자신과 직접 관계된 작은 범위'라는 뜻이죠. 국소적 계산은 결국 전체에서 어떤 일이 벌어지든 상관없이 자신과 관계된 정보만으로 결과를 출력할 수 있다는 것입니다.

국소적 계산

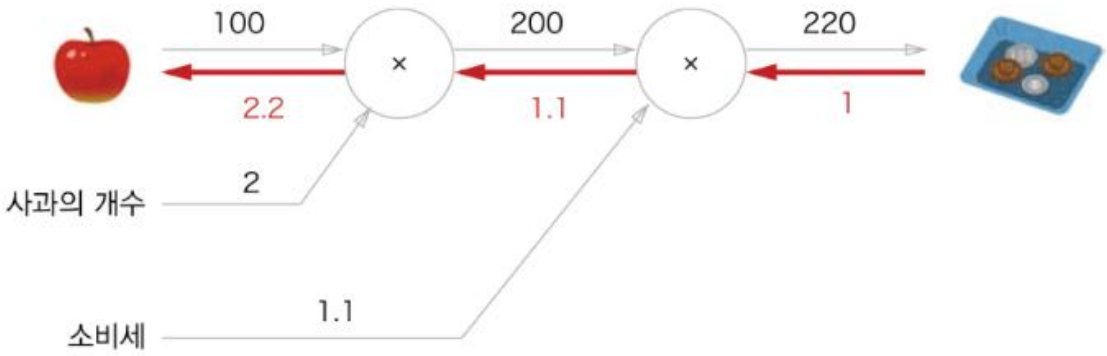
그림 5-4 사과 2개를 포함해 여러 식품을 구입하는 예



[그림 5-4]에서는 여러 식품을 구입하여 (복잡한 계산을 거쳐) 총금액이 4,000원이 되었습니다. 여기에서 핵심은 각 노드에서의 계산은 국소적 계산이라는 점입니다. 가령 사과와 그 외의 물품 값을 더하는 계산(4,000 + 200 → 4,200)은 4,000이라는 숫자가 어떻게 계산되었느냐와는 상관없이, 단지 두 숫자를 더하면 된다는 뜻이죠. 각 노드는 자신과 관련한 계산(이 예에서는 입력된 두 숫자의 덧셈) 외에는 아무것도 신경 쓸 게 없습니다.

국소적 계산

그림 5-5 역전파에 의한 미분 값의 전달



[그림 5-5]와 같이 역전파는 순전파와는 반대 방향의 화살표(굵은 선)로 그립니다. 이 전파는 '국소적 미분'을 전달하고 그 미분 값은 화살표의 아래에 적습니다. 이 예에서 역전파는 오른쪽에서 왼쪽으로 '1 → 1.1 → 2.2' 순으로 미분 값을 전달합니다. 이 결과로부터 '사과 가격에 대한 지불 금액의 미분' 값은 2.2라 할 수 있습니다. 사과가 1원 오르면 최종 금액은 2.2원 오른다는 뜻이죠(정확히는 사과 값이 아주 조금 오르면 최종 금액은 그 아주 작은 값의 2.2배만큼 오른다는 뜻입니다).

국소적 계산

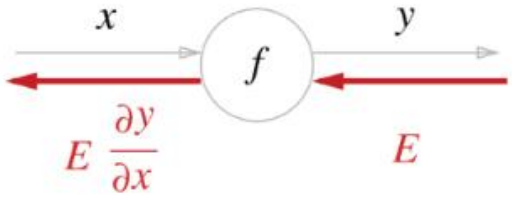
이처럼 계산 그래프는 국소적 계산에 집중합니다. 전체 계산이 제아무리 복잡하더라도 각 단계에서 하는 일은 해당 노드의 '국소적 계산'입니다. 국소적인 계산은 단순하지만, 그 결과를 전달함으로써 전체를 구성하는 복잡한 계산을 해낼 수 있습니다.

NOTE_ 비유하자면 복잡한 자동차 조립은 일반적으로 '조립 라인 작업'에 의한 분업으로 행해집니다. 각 담당자(담당 기계)는 단순화된 일만 수행하며 그 일의 결과가 다음 담당자로 전달되어 최종적으로 차를 완성합니다. 계산 그래프도 복잡한 계산을 '단순하고 국소적 계산'으로 분할하고 조립 라인 작업을 수행하며 계산 결과를 다음 노드로 전달합니다. 복잡한 계산도 분해하면 단순한 계산으로 구성된다는 점은 자동차 조립과 마찬가지인 것이죠.

계산 그래프의 역전파

서둘러 계산 그래프를 사용한 역전파의 예를 하나 살펴봅시다. $y = f(x)$라는 계산의 역전파를 [그림 5-6]로 그려봤습니다.

그림 5-6 계산 그래프의 역전파 : 순방향과는 반대 방향으로 국소적 미분을 곱한다.



[그림 5-6]과 같이 역전파의 계산 절차는 신호 E에 노드의 국소적 미분($\frac{\partial y}{\partial x}$)을 곱한 후 다음 노드로 전달하는 것입니다. 여기에서 말하는 국소적 미분은 순전파 때의 $y = f(x)$ 계산의 미분을 구한다는 것이며, 이는 x에 대한 y의 미분($\frac{\partial y}{\partial x}$)을 구한다는 뜻입니다. 가령 $y = f(x) = x^2$이라면 $\frac{\partial y}{\partial x} = 2x$가 됩니다. 그리고 이 국소적인 미분을 상류에서 전달된 값(이 예에서는 E)에 곱해 앞쪽 노드로 전달하는 것입니다.

연쇄 법칙(Chain rule)

연쇄 법칙을 설명하려면 합성 함수를 알아야 합니다.

합성 함수란 여러 함수로 구성된 함수 입니다.

예를 들어, $z = (x + y)^2$의 식은 다음과 같이 2개의 식으로 구성됩니다.

합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.

$$z = t^2$$

$$t = x + y$$

[식 5.1]

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

[식 5.2]

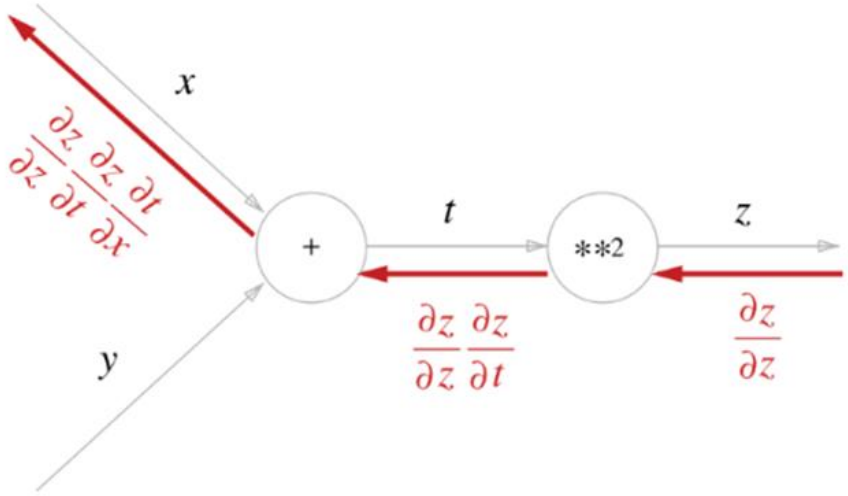
$$\frac{\partial z}{\partial t} = 2t \qquad \frac{\partial t}{\partial x} = 1$$

[식 5.3]

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

[식 5.4]

연쇄 법칙(Chain rule)

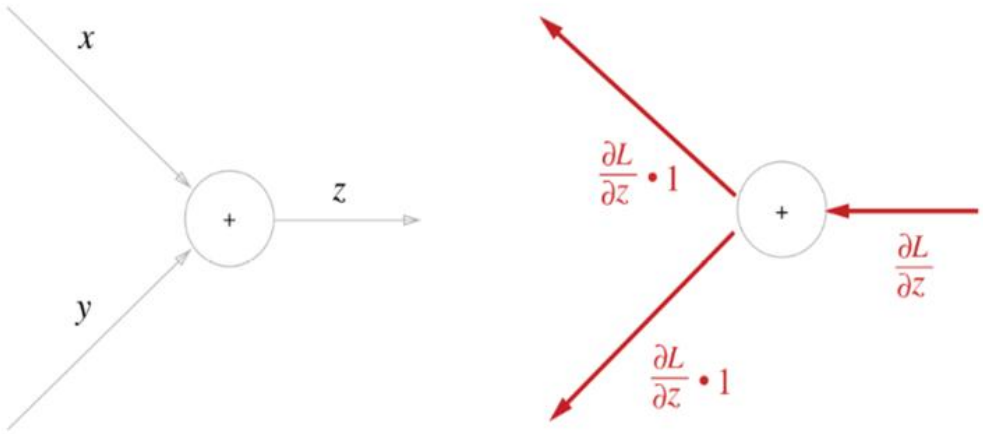


덧셈 노드의 역전파 입니다. 여기에서는 $z = x + y$의 식을 대상으로 역전파를 계산해
보겠습니다.

$z = x + y$의 미분은 다음과 같이 계산 됩니다.

$$\frac{\partial L}{\partial x} = 1, \qquad \frac{\partial L}{\partial y} = 1$$

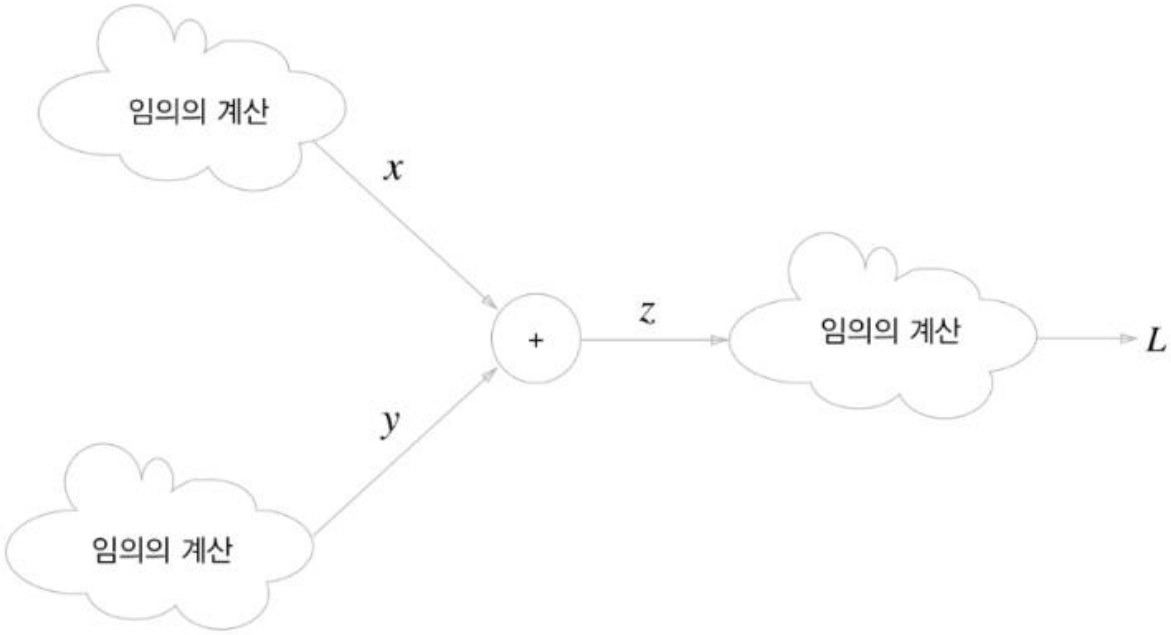
덧셈 노드의 역전파



- 역전파 때는 상류에서 전해진 미분(여기서는 $\frac{\partial L}{\partial z}$)에 1을 곱하고 하류로 흘려 보냅니다.
- 덕셈 노드의 역전파는 1을 곱하기만 할 뿐, 입력된 값을 그대로 다음 노드로 보냅니다.

덧셈 노드의 역전파

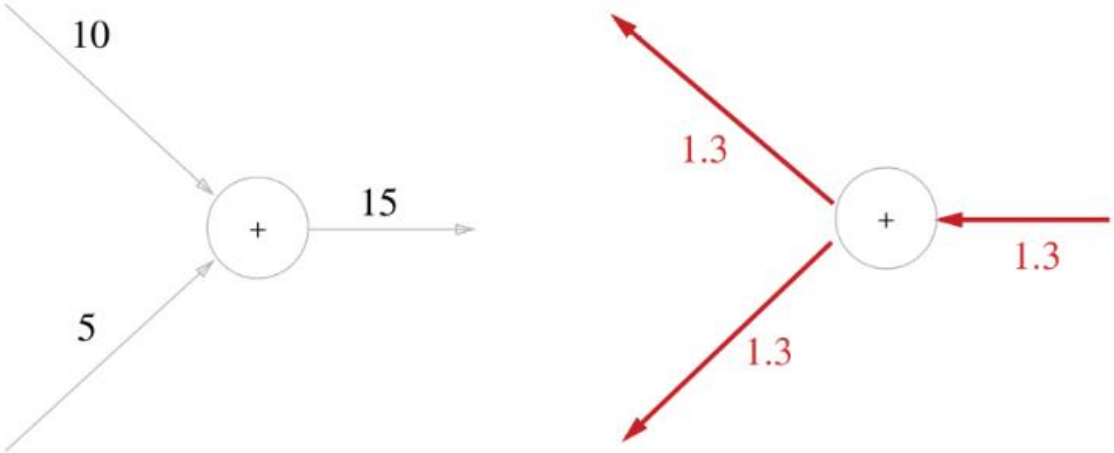
그림 5-10 최종 출력으로 가는 계산의 중간에 덧셈 노드가 존재한다. 역전파에서는 국소적 미분이 가장 오른쪽의 출력에서 시작하여 노드를 타고 역방향(왼쪽)으로 전파된다.



덧셈 노드의 역전파

이제 구체적인 예를 하나 살펴봅시다. 가령 '10 + 5 = 15'라는 계산이 있고, 상류에서 1.3이라는 값이 흘러옵니다. 이를 계산 그래프로 그리면 [그림 5-11]처럼 됩니다.

그림 5-11 덧셈 노드 역전파의 구체적인 예



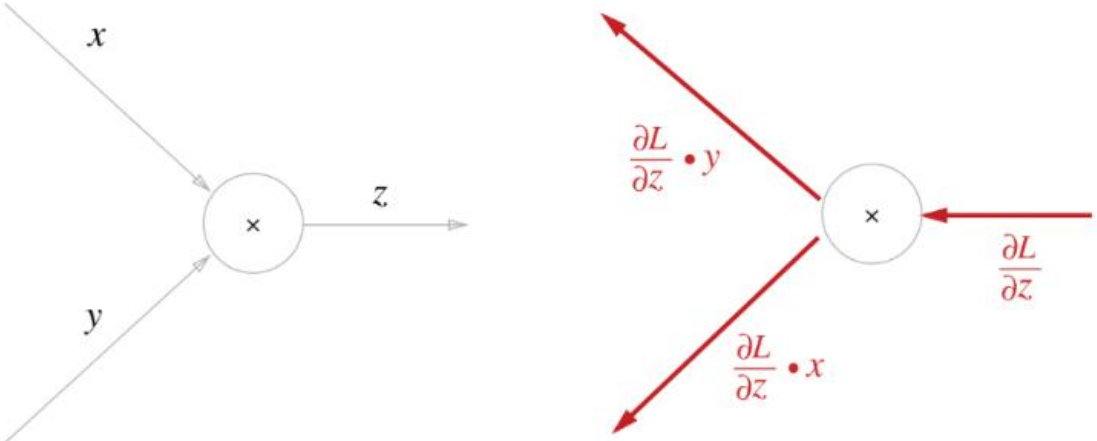
곱셈 노드의 역전파

이어서 곱셈 노드의 역전파를 설명하겠습니다. $z = xy$라는 식을 생각해보죠. 이 식의 미분은 다음과 같습니다.

$$\frac{\partial z}{\partial x} = y$$
$$\frac{\partial z}{\partial y} = x \qquad \text{[식 5.6]}$$

[식 5.6]에서 계산 그래프는 다음과 같이 그릴 수 있습니다.

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

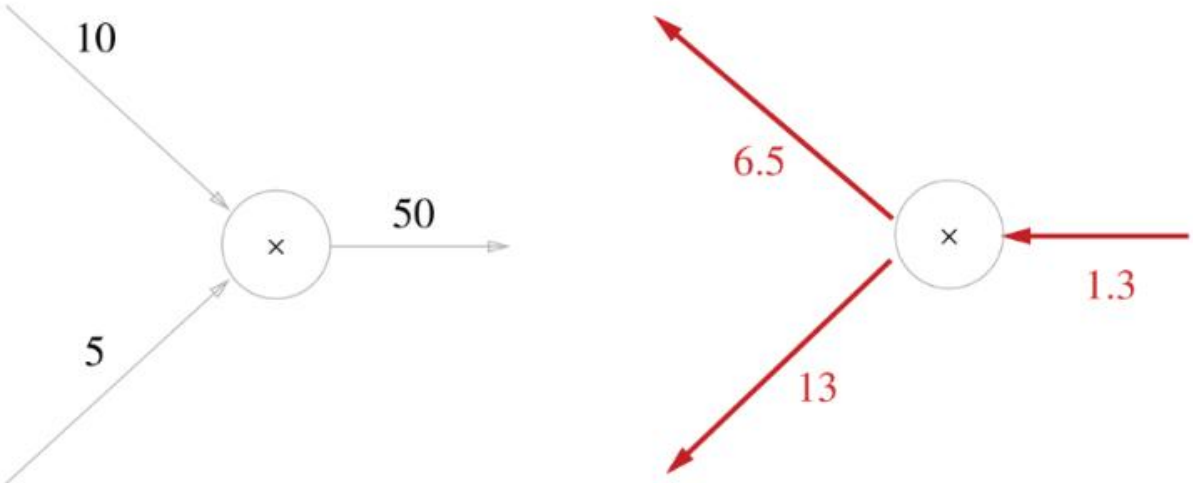


곱셈 노드의 역전파

곱셈 노드 역전파는 상류의 값에 순전파 때의 입력 신호들을 '서로 바꾼 값'을 곱해서 하류로 보냅니다. 서로 바꾼 값이란 [그림 5-12]처럼 순전파 때 x였다면 역전파에서는 y, 순전파 때 y였다면 역전파에서는 x로 바꾼다는 의미입니다.

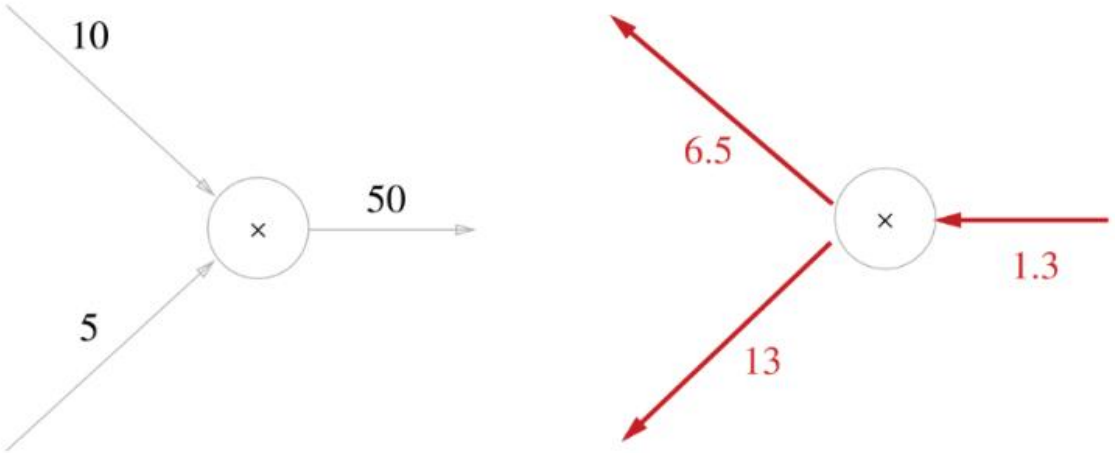
그럼 구체적인 예를 하나 봅시다. 가령 '10 × 5 = 50'이라는 계산이 있고, 역전파 때 상류에서 1.3 값이 흘러온다고 합시다. 이를 계산 그래프로 그리면 [그림 5-13]처럼 됩니다.

그림 5-13 곱셈 노드 역전파의 구체적인 예



곱셈 노드의 역전파

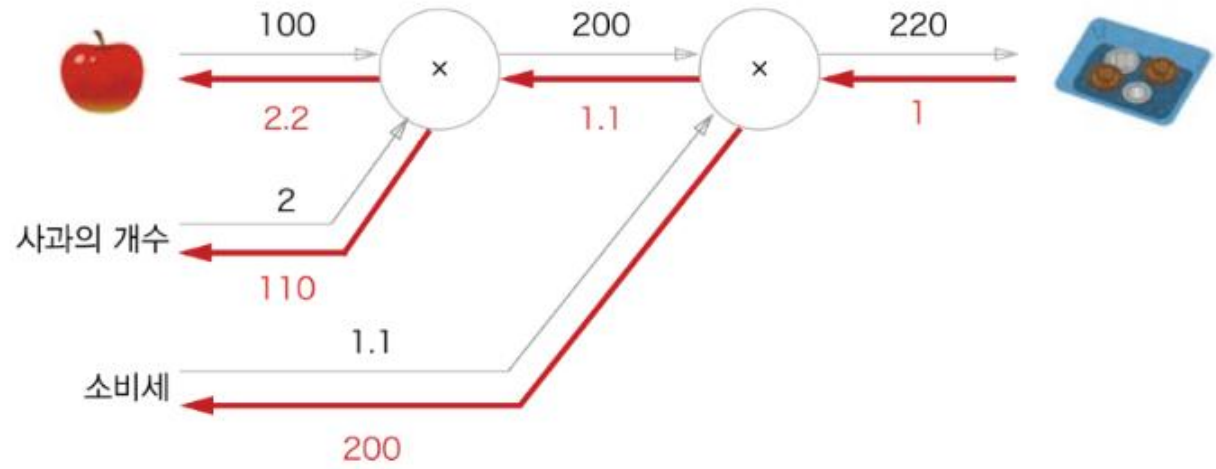
그림 5-13 곱셈 노드 역전파의 구체적인 예



곱셈의 역전파에서는 입력 신호를 바꾼 값을 곱하여 하나는 1.3 × 5 = 6.5, 다른 하나는 1.3 × 10 = 13이 됩니다. 덧셈의 역전파에서는 상류의 값을 그대로 흘려보내서 순방향 입력 신호의 값은 필요하지 않았습니다만, 곱셈의 역전파는 순방향 입력 신호의 값이 필요합니다. 그래서 곱셈 노드를 구현할 때는 순전파의 입력 신호를 변수에 저장해둡니다.

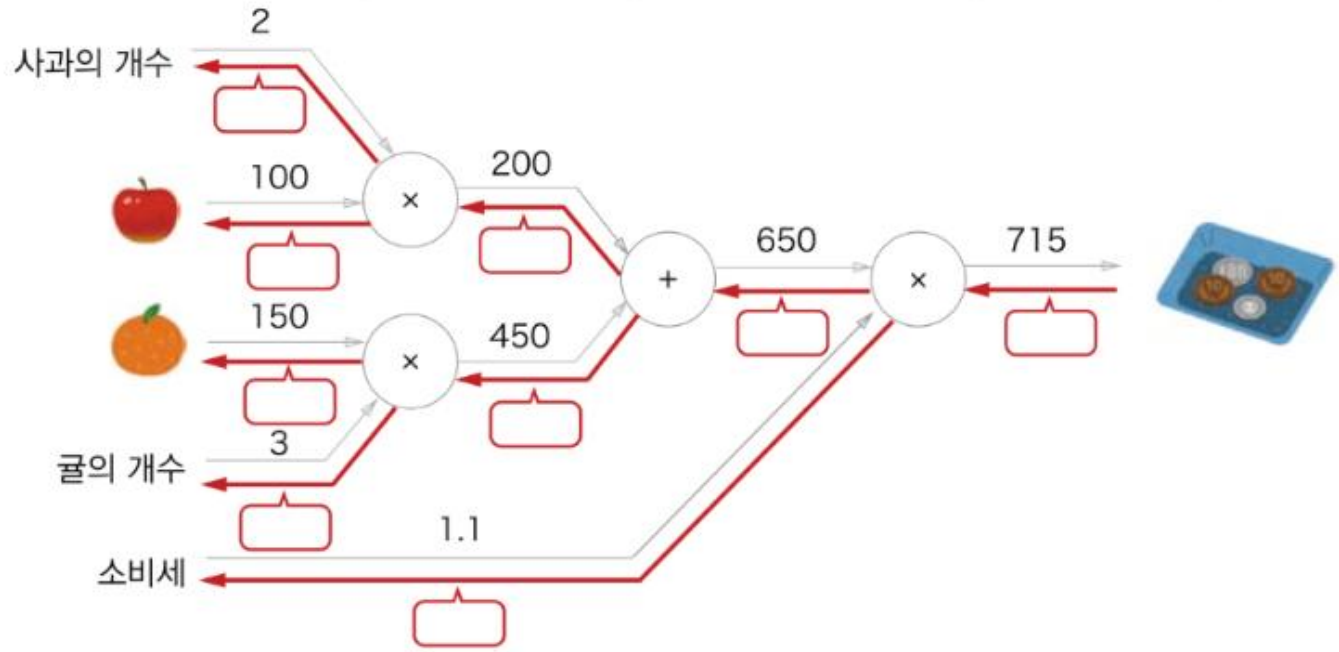
곱셈 노드의 역전파

그림 5-14 사과 쇼핑의 역전파 예



사과와 귤쇼핑의 역전파

그림 5-15 사과와 귤 쇼핑의 역전파 예 : 빈 상자 안에 적절한 숫자를 넣어 역전파를 완성하시오.



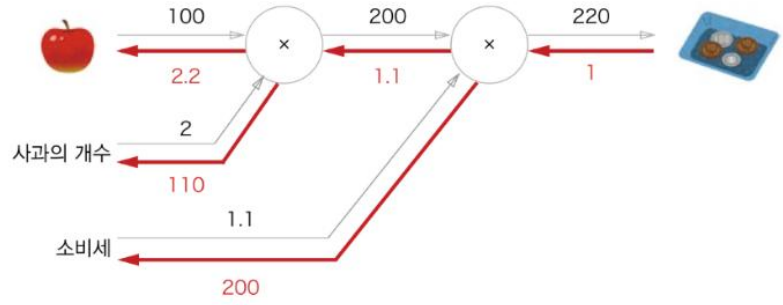
단순한 계층 구현하기

```
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
```

단순한 계층 구현하기

```
apple = 100
apple_num = 2
tax = 1.1

# 계층들
mul_apple_layer = MulLayer()
mul_tax_layer = MulLayer()

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

print(price) # 220
```

그림 5-14 사과 쇼핑의 역전파 예




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

print(dapple, dapple_num, dtax) # 2.2 110 200
```

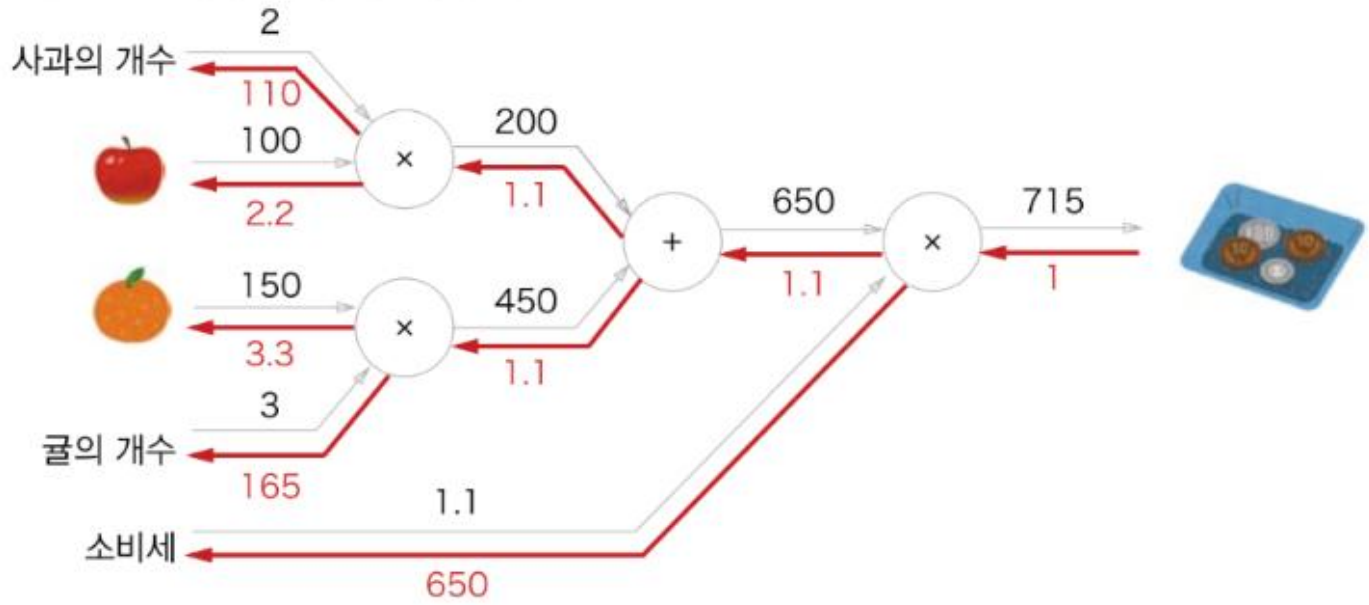
단순한 계층 구현하기

```
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
```

덧셈 계층에서는 초기화가 필요 없으니 __init__()에서는 아무 일도 하지 않습니다(pass가 '아무것도 하지 말라'는 명령입니다). 덧셈 계층의 forward()에서는 입력받은 두 인수 x, y를 더해서 반환합니다. backward()에서는 상류에서 내려온 미분(dout)을 그대로 하류로 흘릴 뿐입니다.

단순한 계층 구현하기

그림 5-17 사과 2개와 귤 3개 구입



단순한 계층 구현하기

ch05/buy_apple_orange.py

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

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

# 순전파
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)

# 역전파
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) # 715
print(dapple_num, dapple, dorange, dorange_num, dtax) # 110 2.2 3.3 165 650
```