오차역전파법 : Backpropagation

[Deep Learning from scratch] - 사이토 고키

중요 내용 요약 : 김진수

오차역전파법 : Backpropagation

- 오차역전파법 (Backpropagation)

  ⇒ **Backward propagation of error**, 오차를 역방향으로 전파하는 방법

- 신경망 학습 시에 **수치 미분**(가중치에 대한 손실 함수의 기울기)을 이용하여 구했었다.

  단순하고 구현이 쉽지만, **계산 시간이 오래 걸린다**.

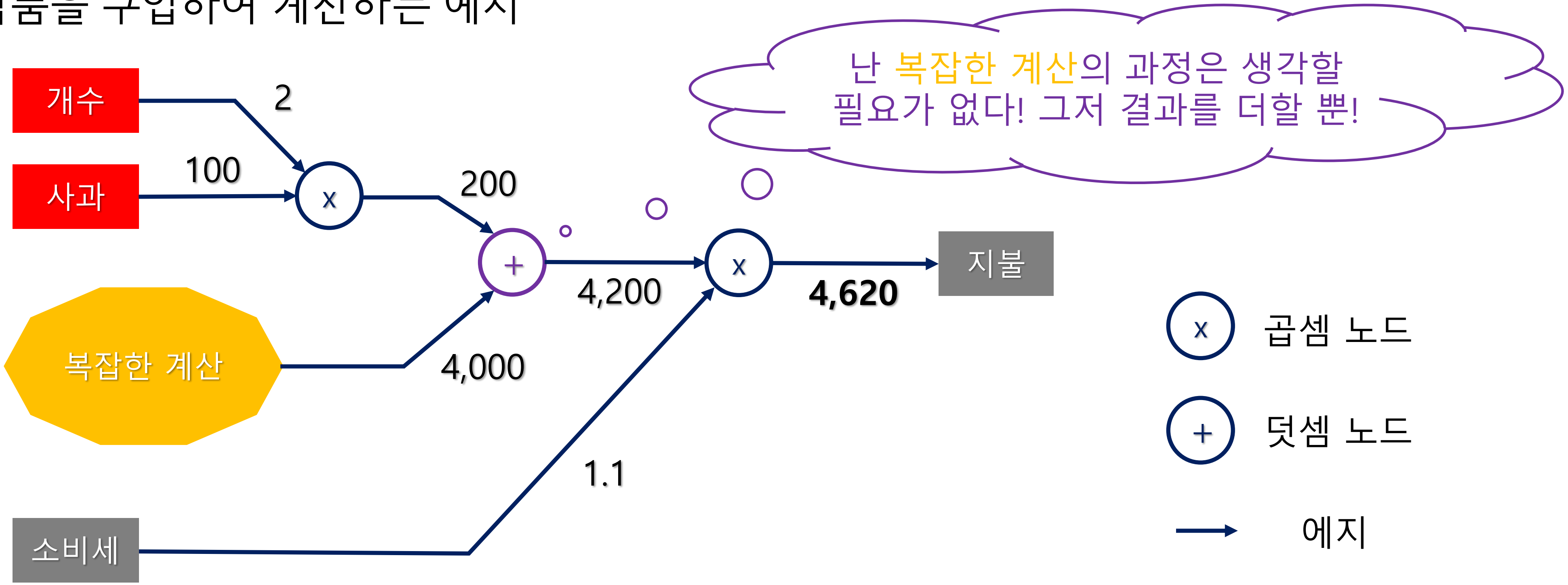
  ⇒ 효율적으로 계산할 수 있는 **오차역전법으로 해결**!

- 오차역전법을 손쉽게 이해하기 위해, **계산 그래프**를 사용

  ⇒ **미분을 효율적으로 계산**하는 과정을 **시각적으로 이해**할 수 있다.

계산 그래프

- 계산 과정을 그래프로 나타낸 것으로, 노드와 에지(노드 사이의 직선)로 표현
- 계산 그래프에서 노드는 자신과 직접 관계된 작은 범위인 국소적 계산을 통해 전파
  ⇒ 전체 계산이 복잡하더라도 **각 노드에서 하는 일은 단순한 국소적 계산의 이점**
- 여러 식품을 구입하여 계산하는 예시
  ⇒



왜 계산 그래프로 푸는가?

- 사과 가격이 오르면 최종 금액에 어떤 영향을 끼치는가?

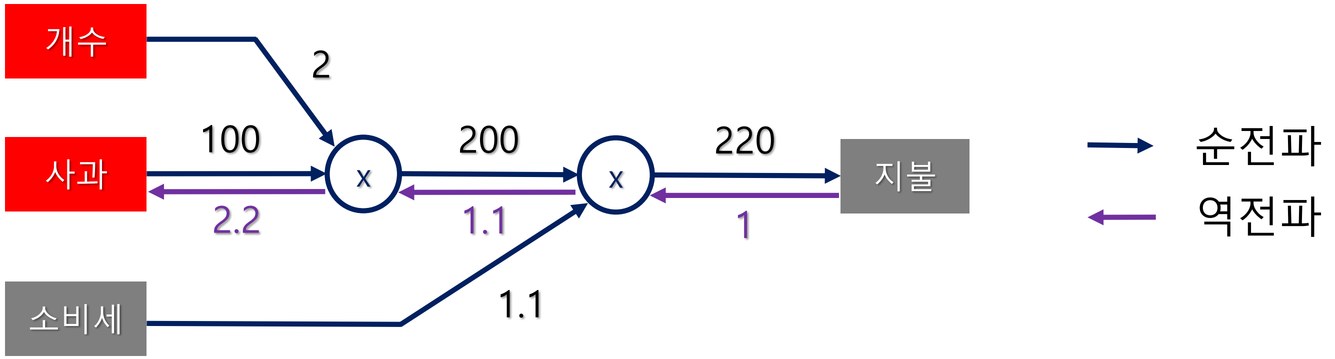
  ⇒ 사과 값이 아주 조금 올랐을 때 지불 금액이 얼마나 증가하는가?

  ⇒ 사과 가격에 대한 지불 금액의 미분 값을 구하는 문제

  ⇒ 계산 그래프에서 역전파를 하면 미분을 효율적으로 계산 가능 (역전파 방법은 뒤에서 설명)

- Ex) 슈퍼에서 사과 1개에 100원인 사과를 2개를 샀을 때의 지불 금액은? 단, 소비세 10% 부과

  ⇒



  ⇒ 반대 방향으로 국소적 미분을 전달하고 그 미분 값은 화살표 아래에 적음

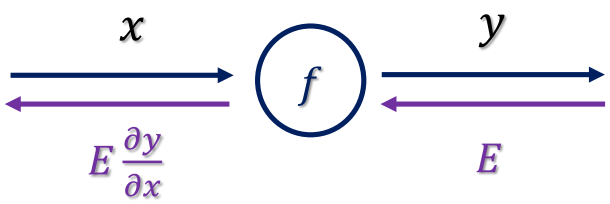
  ⇒ 사과가 1원 오르면 지불 금액은 2.2원 오른다는 뜻

연쇄 법칙 (chain rule)

- $y = f(x)$라는 계산 그래프의 역전파는 연쇄 법칙의 원리로 설명 가능

  ⇒ 신호 E에 노드의 국소적 미분($\frac{\partial y}{\partial x}$)을 곱한 후 다음 노드로 전달하는 것

  ⇒



- 연쇄 법칙은 **합성 함수**(=여러 함수로 구성된 함수)**의 미분에 대한 성질**이며, 합성 함수의 미분은 합성 함수를 구성하는 각 함수의 미분의 곱으로 나타낼 수 있다.

  ⇒ Ex) $z = (x+y)^2$이라는 식은 $z = t^2$, $t = x + y$의 두 개의 식으로 나타낼 수 있다.

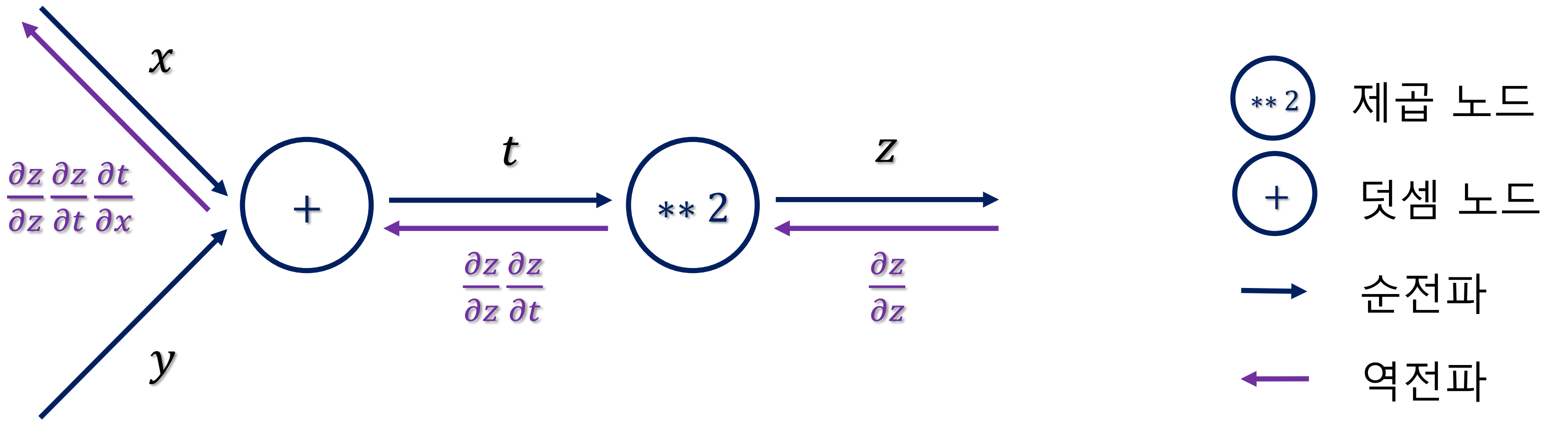
  $z = t^2$를 t에 대해 미분하면 $\boldsymbol{\frac{\partial z}{\partial t}} = \boldsymbol{2t}$이고, $t = x + y$를 x에 대해 미분하면 $\boldsymbol{\frac{\partial t}{\partial x}} = \mathbf{1}$인데,

  구하고 싶은 $\frac{\partial z}{\partial x}$는, 연쇄 법칙을 이용하여 $\boldsymbol{\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x+y)}$으로 나타낼 수 있다.

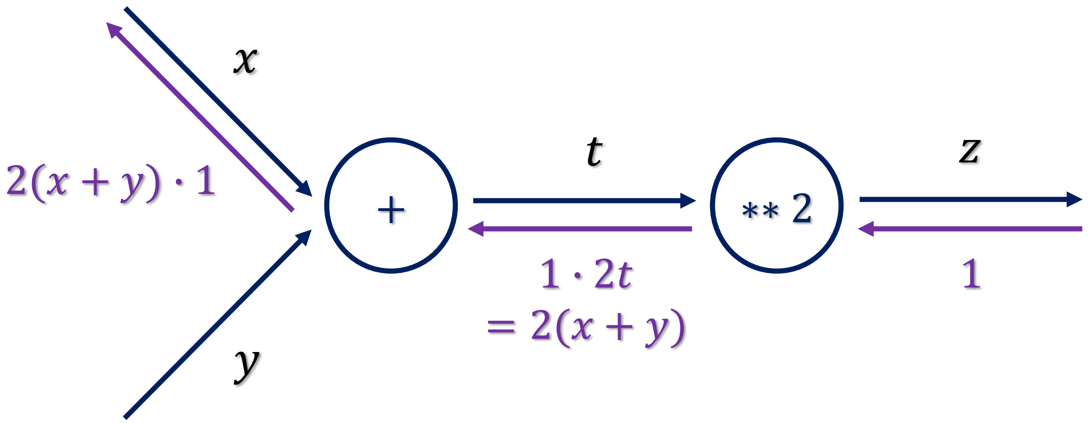
연쇄 법칙 (chain rule)

나타낸 $\frac{\partial z}{\partial t} = 2t$, $\frac{\partial t}{\partial x} = 1$, $\frac{\partial z}{\partial x} = 2(x + y)$를 이용하여 계산 그래프로 나타내면

$\Rightarrow$



$\Rightarrow$



역전파가 하는 일은 연쇄 법칙의

원리와 같이 $\frac{\partial z}{\partial x} = 2(x + y)$이 됨을

알 수 있다.

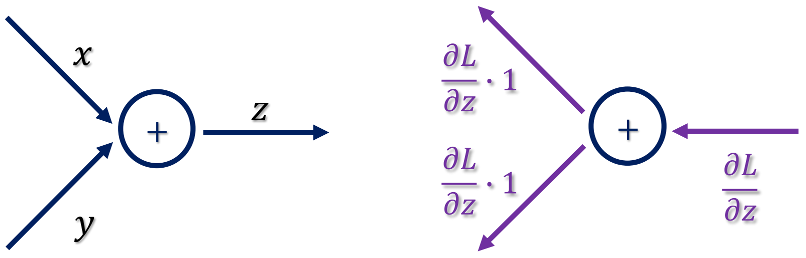
역전파 (Backward propagation)

- 덧셈 노드의 역전파 ($z = x + y$ 식으로 설명, x와 y에 대한 미분 식은 각각 $\frac{\partial z}{\partial x} = 1$, $\frac{\partial z}{\partial y} = 1$이고

  여기서 L은 이후 전체 계산 그래프의 최종 출력 값을 가정)

  ⇒ 상류에서 전해진 미분($\frac{\partial L}{\partial z}$)에 1을 곱하여 하류로 보낸다.

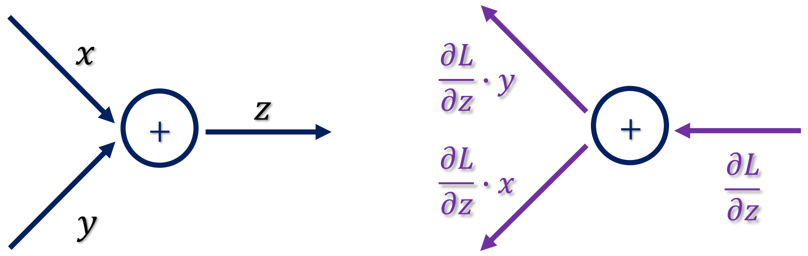
  즉, 입력된 값을 그대로 다음 노드로 보내게 된다.



- 곱셈 노드의 역전파 ($z = xy$ 식으로 설명, x와 y에 대한 미분 식은 각각 $\frac{\partial z}{\partial x} = y$, $\frac{\partial z}{\partial y} = x$이고

  여기서 L은 이후 전체 계산 그래프의 최종 출력 값을 가정)

  ⇒ (상류 값) * (순전파의 서로 바꾼 값)을 하류로 보낸다.

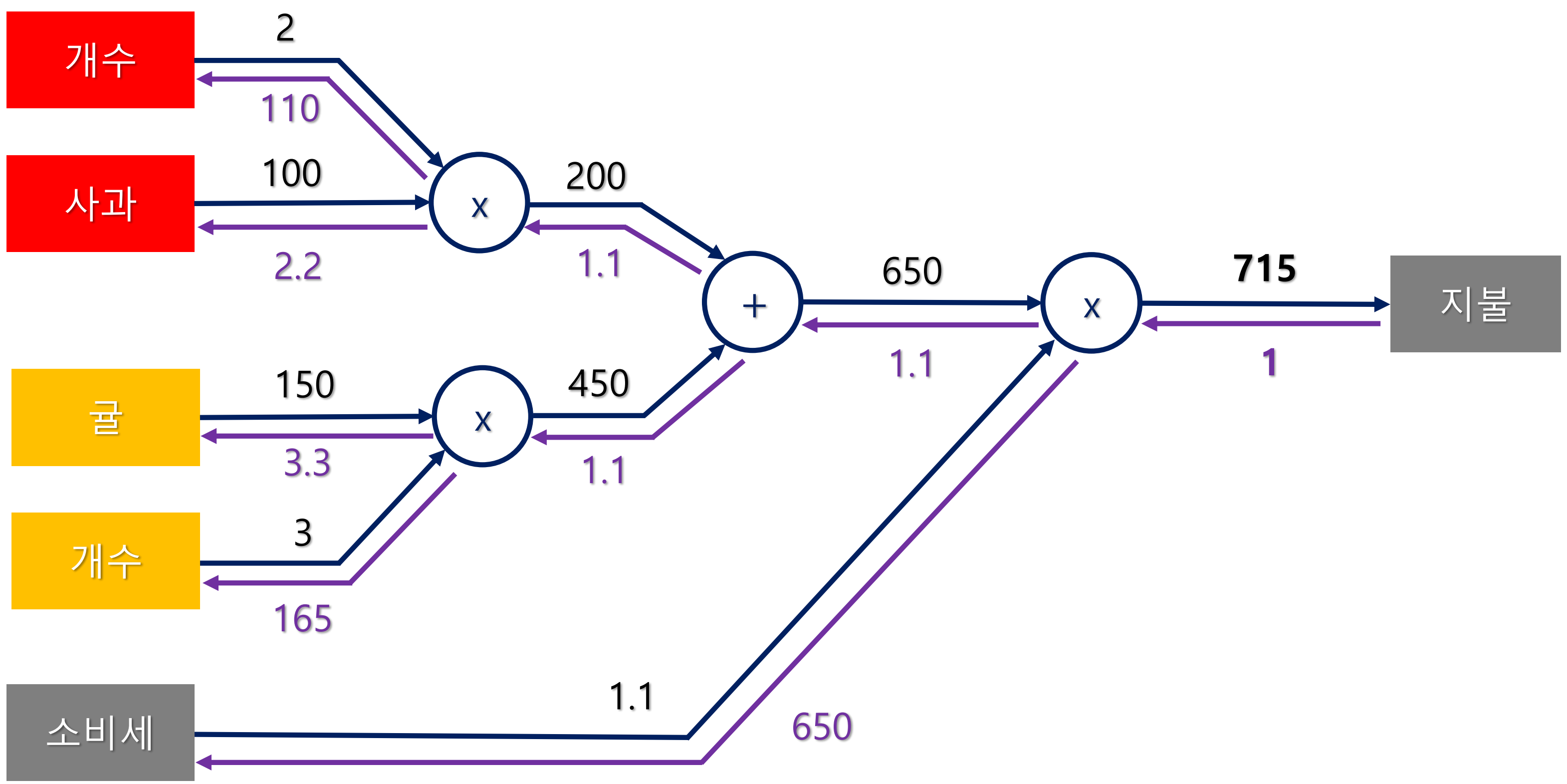


역전파 (Backward propagation)

- Ex) 1개에 100원인 사과 2개와 1개에 150원인 귤 3개를 살 때의 가격을 구하는 계산 그래프에서,

  소비세는 10%일 때, 각 변수의 미분을 구해 역전파를 완성하라.

  ⇒



역전파 단순한 계층 구현 : Code

```
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
        dy = dout * self.x
        return dx, dy

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
```

- $MulLayer$라는 class를 생성하여 곱셈 노드를 만들어 준다.

  $_init_$에서 $self.x$ 와 $self.y$를 생성해주고, 선언만 해놓는다.

  $\Rightarrow$ $backward$ 시에 순전파 값을 서로 바꾸어 곱해주기 위한 변수

  $forward$에서는 두 입력을 곱하여 return 해준다.

  $Backward$에서는 상류의 값 $*$ 순전파의 서로 바꾼 값을 return 해준다.

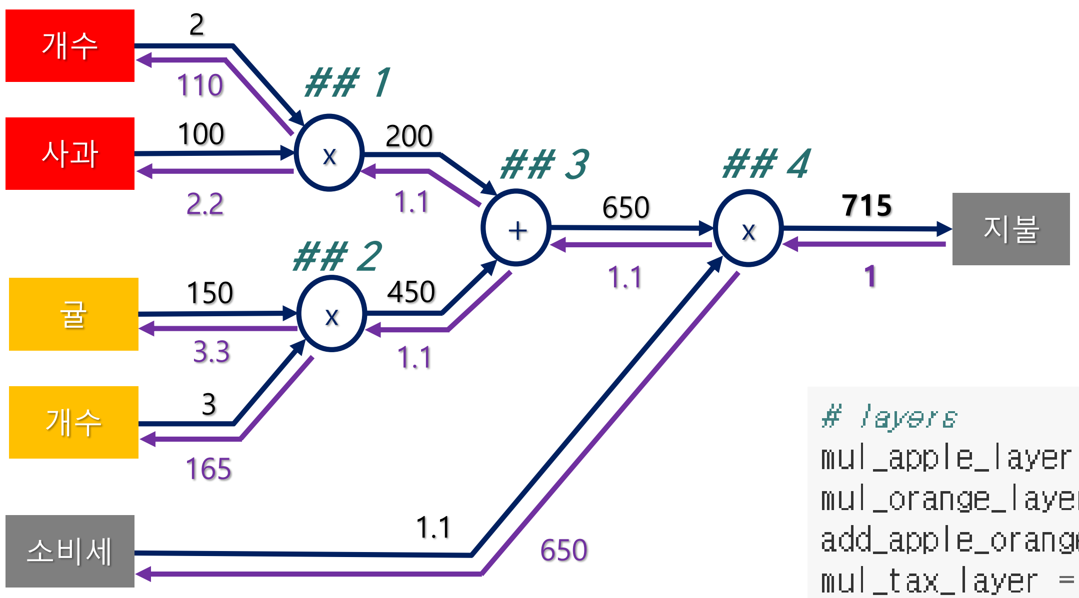
- AddLayer라는 class를 생성하여 덧셈 노드를 만들어 준다.

  $_init_$에서 기억해줄 변수가 없으므로 $pass$

  $forward$에서는 두 입력을 더하여 return 해준다.

  $Backward$에서는 상류의 값을 그대로 두 변수에 return 해준다.

역전파 단순한 계층 구현 : Code



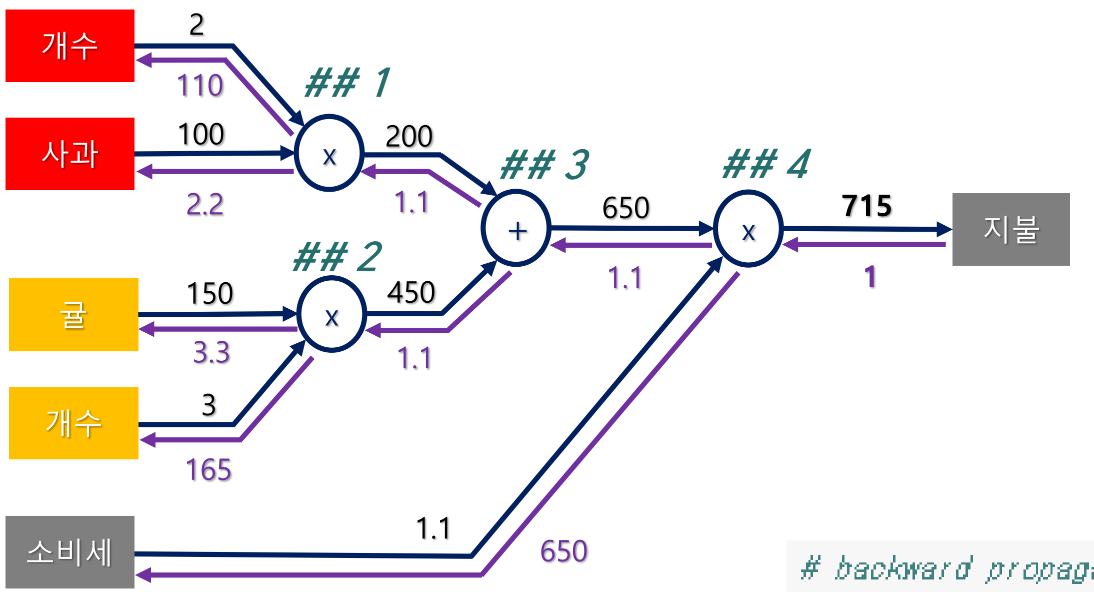

```
# inputs
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1
```

```
# layers
mul_apple_layer = MulLayer()
mul_orange_layer = MulLayer()
add_apple_orange_layer = AddLayer()
mul_tax_layer = MulLayer()
```

```
# forward propagation
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
```

- 사과 값 & 개수, 오렌지 값 & 개수, 소비세를 선언한다.
- 사과 값과 개수를 곱해줄 곱셈 노드, 오렌지 값과 개수를 곱해줄 곱셈 노드, 사과와 오렌지 값을 합쳐줄 덧셈 노드, 소비세를 곱해줄 곱셈 노드를 선언한다.
- 순전파를 위해 각 노드마다 $forward$를 진행한다. 여기서, *## 1 ~ ## 4* 순서에 맞게 진행한다.

역전파 단순한 계층 구현 : Code



- 역전파를 위해 각 노드마다 *backward*를 진행한다.

  여기서, *## 4* ~ *## 1* 순서에 맞게 진행한다.

- 입력 방향으로 역전파된 값들을 *print* 해보면,

  미분으로 구했던 것과 같은 결과를 확인할 수 있다.

```python
# backward propagation
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
```

```python
print(price)
print(dapple_num, dapple, dorange_num, dorange, dtax)
```

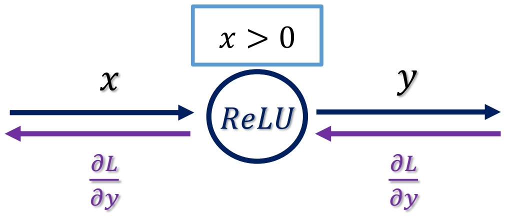
```
715.0000000000001
110.00000000000001 2.2 165.0 3.3000000000000003 650
```

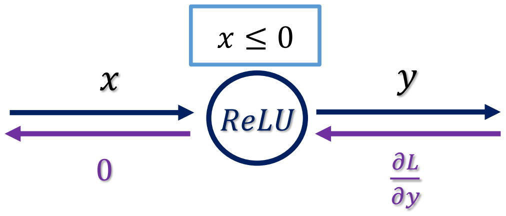
활성화 함수 계층

- ReLU 계층

$$y = \begin{cases} 0 \ (x \le 0) \\ x \ (x > 0) \end{cases}$$ 에서 x에 대한 y의 미분에 관한 식으로 정리하면, $$\frac{\partial y}{\partial x} = \begin{cases} 0 \ (x \le 0) \\ 1 \ (x > 0) \end{cases}$$

⇒ 계산 그래프로는,





- Sigmoid 계층

$$y = \frac{1}{1+\exp(-x)}$$ 에서 에서 x에 대한 y의 미분에 관한 식으로 정리하면, $$\frac{\partial y}{\partial x} = \frac{\exp(-x)}{(1+\exp(-x))^2} = \underline{y(1-y)}$$

⇒ 계산 그래프로는,

Sigmoid 계층의 역전파(미분 값)는
순전파의 출력(y)만으로 계산할 수 있는 특징이 있다.

활성화 함수 계층 : Code

```python
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
```

$numpy$에서 $out = x$와 같은 명령어를 사용하면 효율적으로 배열을 처리하기 위해 메모리를 추가로 잡지 않고 x의 데이터와 속성을 공유하므로 x를 바꾸면 out도 바뀐다. 이럴 때 $copy()$를 이용하면 또다른 메모리 공간에 값만 같은 배열을 복사한다.

```python
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 * self.out * (1.0 - self.out)
        return dx
```

- $Relu$ 계층에서는 $\_init\_$ 시에 $self.mask$ 라는 변수를 만들어서 $x <= 0$ 인 경우의 값을 0으로 처리하는 용도로 사용한다.
  $forward$ 에서 받은 x를 이용하여 $self.mask$ 필터를 만든다.
  ⇒ $numpy$에서의 조건문으로 $True$ / $False$의 배열을 만들어 낸다.
  $x.copy()$를 이용하여 out이라는 새로운 변수를 만들고 $self.mask$를 이용하여 필터링한 부분들을 0으로 바꿔준 후에 return 해준다.
  $backward$ 시에 $dout$에 $self.mask$를 적용한 변수인 dx를 return 해준다.
- Sigmoid 계층에서는 __init__ 시에 self.out 라는 변수를 만들어서 forward 시에 출력 변수를 저장해 놓았다가, backward 시에 변수로 사용한다.
  $forward$ 에서 받은 x를 이용하여 출력시에 $self.out$에 out을 저장하며 return
  backward 시에 Sigmoid의 미분 값은 $\frac{\partial y}{\partial x} = y(1-y)$인 점을 이용하여 dx를 return

어파인 계층 (Affine Layer)

- 신경망에서 **행렬 계산을 수행하는 계층을 어파인 계층(Affine Layer)**이라 한다.

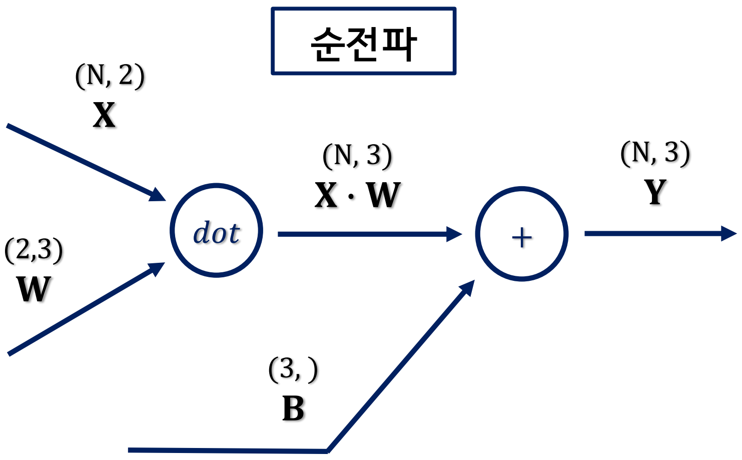
  ⇒ 행렬의 곱을 기하학에서 어파인 변환(Affine transformation)이라 한다.

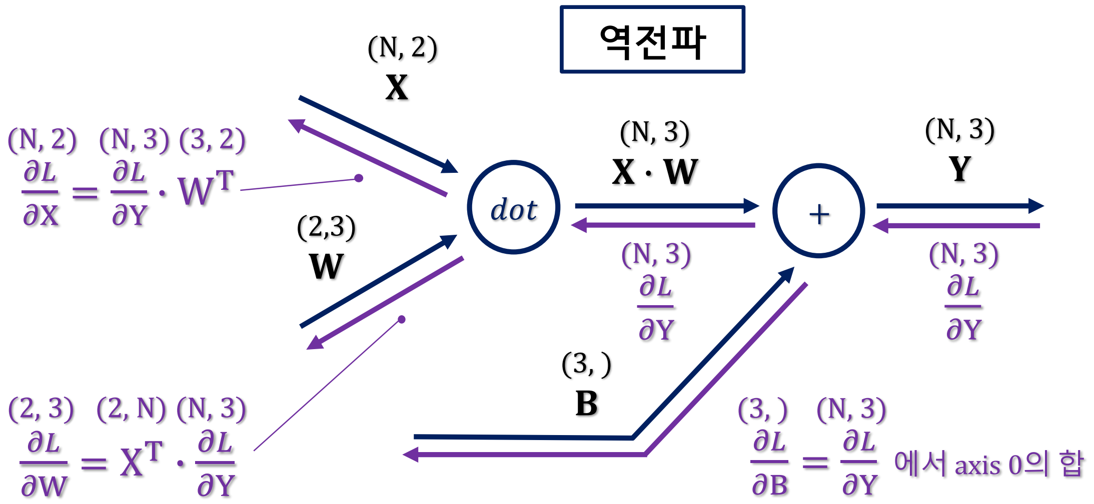
  ⇒ 다차원 배열 계산 때 사용했던 방법인 $\boldsymbol{np.dot(X, W)} + \boldsymbol{B}$의 계산 그래프

  ⇒ 이전과는 다르게 계산 그래프에 '스칼라 값'이 아닌 '**행렬**'이 흐르고 있다.

  ⇒ Ex) 데이터 N개인 X, W, B 가 행렬(다차원 배열)임에 유의, shape을 위에 표기했다.

  여기서, $\frac{\partial L}{\partial \mathrm{W}} = \mathrm{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathrm{Y}}$ 이고, $\frac{\partial L}{\partial \mathrm{X}} = \frac{\partial L}{\partial \mathrm{Y}} \cdot \mathrm{W}^{\mathrm{T}}$이다. (⇒ $\mathrm{X} \cdot \mathrm{W} = \mathrm{W}^{\mathrm{T}}\mathrm{X} = \mathrm{X}^{\mathrm{T}}\mathrm{W}$ 성질 이용, T는 전치행렬)





어파인 계층 (Affine Layer) : Code

```python
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
```

- $Affine$ 계층에서는 $\_\_init\_\_$ 시에 W, b를 각각 $self.W, self.b$의 변수로 받고, $self.x, self.dW, self.db$는 선언만 해놓는다.
- $forward$에서 입력 x와 $self.W$를 행렬 곱한 후, $self.b$를 더하여 return
  ⇒ 식으로 나타낸다면, $\mathbf{Y} = \mathbf{X} \cdot \mathbf{W} + \mathbf{B}$에서 $\mathbf{Y}$를 return
- $backward$에서는 역전파를 위해 출력층 쪽에서 넘어온 $self.W.T$ ($self.W$의 전치행렬)에 $dout$을 행렬 곱해준 dx를 return 해준다.

  ⇒ $\frac{\partial L}{\partial \mathrm{X}} = \frac{\partial L}{\partial \mathrm{Y}} \cdot \mathrm{W}^{\mathrm{T}}$이므로 $dx(\frac{\partial L}{\partial \mathrm{X}})$는 $dout(\frac{\partial L}{\partial \mathrm{Y}})$과 $self.W.T(\mathrm{W}^{\mathrm{T}})$의 행렬 곱($np.dot$)이고,

  $\frac{\partial L}{\partial \mathrm{W}} = \mathrm{X}^{\mathrm{T}} \cdot \frac{\partial L}{\partial \mathrm{Y}}$ 이므로 $self.dW(\frac{\partial L}{\partial \mathrm{W}})$는 $self.x.T(\mathrm{X}^{\mathrm{T}})$과 $dout(\frac{\partial L}{\partial \mathrm{Y}})$의 행렬 곱($np.dot$)이다.

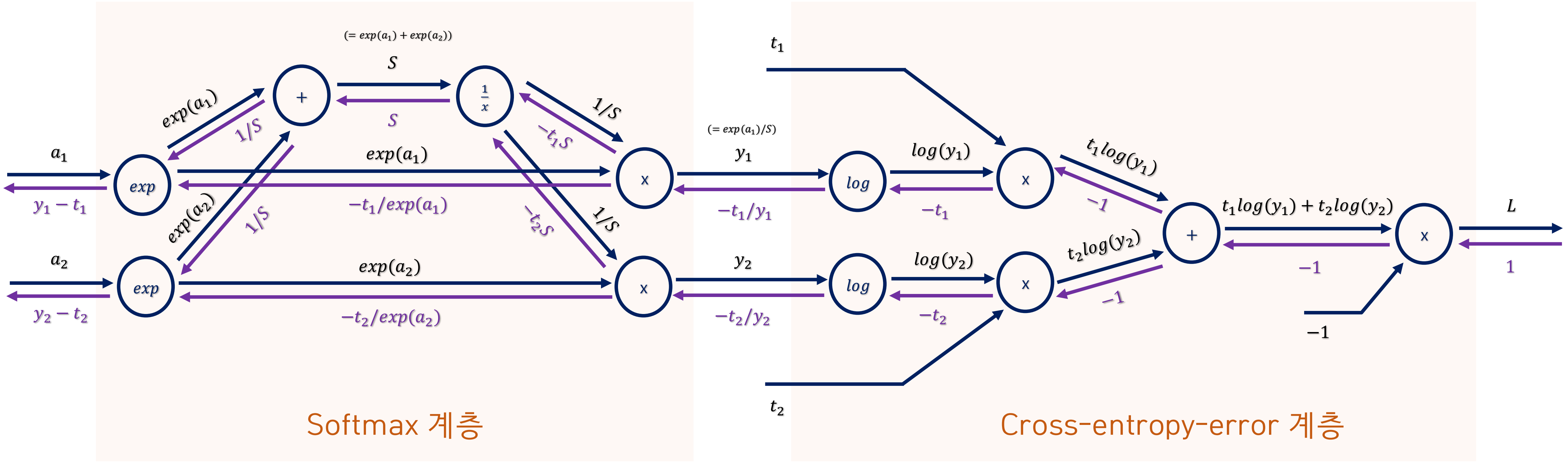
  $\frac{\partial L}{\partial \mathrm{B}} = \frac{\partial L}{\partial \mathrm{Y}}$ 에서의 axis 0의 합이므로 $self.db$는 $dout(\frac{\partial L}{\partial \mathrm{Y}})$의 $np.sum$이고 $axis = 0$ 기준이다.

Softmax-with-Loss 계층

- Softmax-with-Loss 계층의 계산 그래프

⇒ Softmax의 식 $y_k = \frac{exp(a_k)}{\sum_{i=1}^{n} exp(a_i)}$ 값을 넘겨준 Cross-entropy-error(Loss)의 식 $L = -\sum_k^K t_k log y_k$의 계산 그래프

여기선 a_k, y_k, t_k의 개수 $K = 2$개인 경우로 가정하여 계산했다.

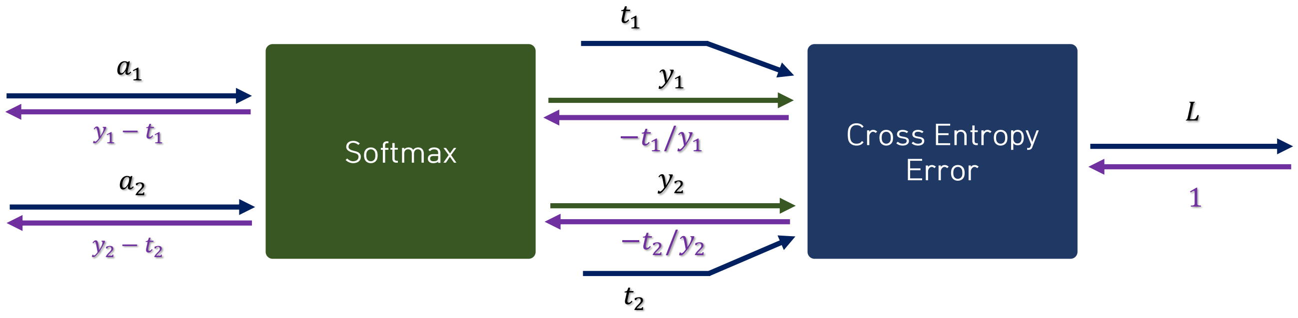


Softmax-with-Loss 계층

- 간소화한 Softmax-with-Loss 계층의 계산 그래프

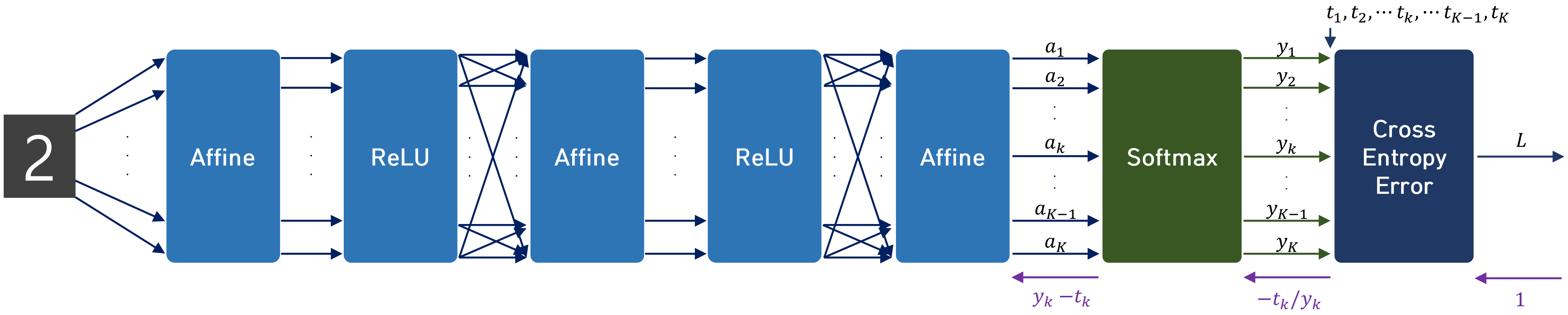
  ⇒ Softmax의 손실 함수로 Cross-entropy-error는 역전파가 $(y_k - t_k)$로 깔끔하게 떨어지게 설계 되어있습니다.

  ⇒



- MNIST 손글씨 숫자 인식 신경망에서의 Softmax-with-Loss 계층 예시

  ⇒ 입력층(숫자 이미지)에서 Affine 계층과 활성함수 계층(ReLU)들을 지나서 출력층(Softmax)을 거친 결과(y_k)를 이용하여 정답 레이블(t_k)과의 손실 함수(Cross-entropy-error)를 구한 후, 역전파를 통해 신경망을 학습한다.



Softmax-with-Loss 계층 : Code

```python
class SoftmaxWithLoss:
    def __init__(self):
        self.loss = None
        self.y = None  # softmax 출력
        self.t = None  # 정답 레이블(one-hot)

    def forward(self, x, t):
        self.t = t
        self.y = softmax(x)
        self.loss = batch_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
```

- $SoftmaxWithLoss$ 계층에서는 $\_init\_$ 시에 최종 loss인 $self.loss$, softmax의 출력인 $self.y$, 정답 레이블인 $self.t$를 생성한다.
- $forward$ 시에 입력 x를 받아서 $softmax$에 통과시킨 결과인 $self.y$를 구한다. $self.y$ 와 정답 레이블인 $self.t$를 이용하여 $batch_cross_entropy_error$를 구하면, $self.loss$를 얻을 수 있고, 그 값을 return한다.
- $backward$ 시에 먼저 $self.t.shape[0]$를 $batch_size$로 받아서 데이터 개수만큼 나눠줄 변수를 받아 놓는다. 역전파가 $(y_k - t_k)$로 깔끔하게 떨어지므로, return 해줄 dx는 $self.y - self.t$를 $batch_size$로 나눠준 값이 된다.

오차역전파법을 적용한 신경망 구현

신경망의 매개변수(가중치와 편향)를 MNIST 훈련 데이터로 조정하는 '신경망 학습'을 진행

- 1단계 : 미니배치

  ⇒ 훈련 데이터의 미니배치를 무작위로 선정하는 확률적 경사 하강법(Stochastic gradient descent : SGD)를 이용하여 손실 함수 값을 줄이는 것이 목표

- 2단계 : 기울기 산출 ⇒ **기울기 산출 시, 수치 미분 방법을 오차역전파법으로 변경!**

  ⇒ 미니배치의 손실 함수 값을 줄이기 위해 가중치 매개변수의 기울기를 구하여, 손실 함수의 값을 가장 작게 하는 방향을 제시

- 3단계 : 매개변수 갱신

  ⇒ 가중치 매개변수를 기울기 방향으로 아주 조금 갱신 (학습률에 비례)

- 4단계 : 반복

  ⇒ 1~3 단계를 반복

오차역전파법을 적용한 신경망 구현 : Code

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size,
                 weight_init_std=0.01):
        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)

        # layer - Affine, ReLU
        # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        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()
        # >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    def predict(self, x):
        # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
        for layer in self.layers.values():
            x = layer.forward(x)
        # >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
        return x

    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)
```

- 앞서 구현된 Code에서 추가된 것들
  ⇒ *Softmax 함수 : 입출력이 2차원 배열일 때 행렬 변환 계산
  *Affine 계층 : 텐서 대응 reshape
  *SoftmaxWithLoss 계층 : 정답 레이블이 one-hot 인코딩 아닐 경우
  *(batch)Cross_entropy_error 계층 : 정답 레이블이 one-hot 인코딩일 경우

- 주석 # <…< 와 # >…>는 이번 장의 내용이므로 집중해서 봐야하는 부분이다.

- $TwoLayerNet$ class의 _init_ 시에, 마지막 계층인 $self.lastLayer$는 $SoftmaxWithLoss$, $Affine$과 $ReLU$ 계층을 의미하는 $self.layers$ 추가하고 이외의 부분은 동일하다.
  ⇒ $self.layers$는 $OrderedDict$ 함수를 이용하여 순서가 있는 딕셔너리를 만들고, 순차적으로 $Affine1,\ Relu1,\ Affine2$를 $self.layers$에 넣어준다.

- $predict$ 시에는 $self.layers$의 value들을 차례대로 불러와서 $forward$ 시켜주고, 모든 $layer$들을 거친 값인 x를 return 해준다.
  ⇒ $(layer)\ Affine1,\ Relu1,\ Affine2$를 순서대로 거치는 과정

- $loss$ 에서는 x를 self.$predict(= self.layers$의 $forward)$시킨 y와 t를 $self.lastLayer.forward$ $(= SoftmaxWithLoss)$ 에 대입한 값을 return

오차역전파법을 적용한 신경망 구현 : Code

```python
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):
    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

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() # reverse ordered dict list
    for layer in layers:
        dout = layer.backward(dout)
    # >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    grads = {}
    grads['W1'] = self.layers['Affine1'].dW
    grads['b1'] = self.layers['Affine1'].db
    grads['W2'] = self.layers['Affine2'].dW
    grads['b2'] = self.layers['Affine2'].db
    return grads
```

- $accuracy$에서는 입력 x를 $self.predict$ 에 대입한 결과 y 에서 가장 큰 값을 np.$argmax$로

  ($axis = 1,\ MNIST$에선 10개 중 1개) 골라내고, 정답 레이블 t와 비교하여 정확도를 return

- $(self)numerical_gradient$는 수치 미분을 이용하여 신경망의 기울기를 구하는 방법이다.

  ⇒ 앞선 신경망 학습 장에서 설명했으므로 생략, 오차역전파법과 비교하기 위한 지표이다.

- $gradient$는 오차역전파법을 적용한 신경망의 기울기를 구하는 방법이다.

  먼저, 비교를 위해 $self.loss$로써 다시 $forward$를 진행한 후에, $dout = 1$부터 시작하여

  $self.lastLayer.backward(= SoftmaxWithLoss.backward)$를 거친 값을 $dout$에 덮어쓴다.

  순서가 있는 딕셔너리 $self.layers$는 $list$와 $reverse$를 활용하여 반대 순서로 재가공해준다.

  for문을 돌리면서 $layer$마다 $dout$을 이용하여 $backward$ 시켜준다.

  기울기를 구하기 위해서, $self.layers$의 '$Affine1$'에 대한 미분(역전파) 값은 '$W1$','$b1$' 값으로,

  '$Affine2$'에 대한 미분 값은 '$W2$','$b2$' 값으로 $grads$에 저장하여 return 해준다.

오차역전파법으로 구한 기울기 검증 : Code

```python
(x_train, t_train), (x_test, t_test) = \
        load_mnist(normalize=True, one_hot_label=True)
x_batch = x_train[:10]
t_batch = t_train[:10]
```

```python
network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)
grad_numerical = network.numerical_gradient(x_batch, t_batch)
grad_backprop = network.gradient(x_batch, t_batch)
```

```python
for key in grad_numerical.keys():
    diff = np.average(np.abs(grad_backprop[key] - grad_numerical[key]))
    print(key + ":" + str(diff))
```

```
b1:1.702692901953606e-09
b2:6.033662941842821e-08
W2:3.125925059650823e-09
W1:2.6642440043640307e-10
```

- 먼저 MNIST를 load해서, 입력과 정답을 10개씩만 batch로 떼어놓았다. 그 후에 만들어 놓은 $TwoLayerNet$ class에 적당한 사이즈를 입력하여 $network$를 생성하였다.
- $grad_numerical$은 수치 미분을 이용한 방식으로, 구현이 간단하고 정확하다.
  ⇒ 정확한 계산을 하는 지에 대한 오차역전파법 방식과의 비교 검증
- $grad_backprop$은 오차역전파법을 이용한 방식으로, 구현은 수치 미분 방식에 비해 상대적으로 복잡하지만 계산 시간이 빠르고 효율적인 장점이 있다.
  ⇒ 정확도를 비교하였을 때, 거의 차이가 없다고 판단되면 훌륭한 방법!
- $grad_numerical$과 $grad_backprop$에서의 같은 key에서의 오차 값들의 평균을 내어서 비교했을 때, 그 값이 매우 작다면 올바르게 검증되었다고 할 수 있다.
  ⇒ 평균적으로 모든 변수들에게서 10^{-7} 이하의 오차 값이 나오므로, 수치 미분의 값과 거의 같으며 올바른 값이라고 말할 수 있다.

오차역전파법을 사용한 학습 구현 : Code

```
(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)
```

```
# 수치 미분 때와 동일하게
iters_num = 1000
train_size = x_train.shape[0]
batch_size = 500  # 실제 코드 : 100
learning_rate = 0.1

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

iter_per_epoch = train_size // batch_size
print('iter_per_epoch =', iter_per_epoch)
print('epochs =', iters_num // iter_per_epoch
      if iters_num % iter_per_epoch == 0 else iters_num // iter_per_epoch + 1)
```

```
iter_per_epoch = 120
epochs = 9
```

- MNIST를 load한 후에 만들어 놓은 $TwoLayerNet$에 필요한 값들을 입력하여 $network$를 생성하였다.

  ⇒ 비교를 위해 앞선 수치 미분을 적용한 학습 구현 때와 동일한 값들을 사용

- $iters\_num$, $train\_size$, $batch\_size$ ⋯ 등등 여러 변수들 역시 앞선 수치 미분을 적용한 학습 구현 때와 동일한 값들을 사용

  ⇒ 동일한 값들을 기준으로 학습 시간, 정확도의 차이를 비교

- $iter\_per\_epoch$ (훈련용 데이터 크기 / 배치 사이즈)을 반복 횟수로 나누어준 값, 즉, 훈련 데이터 전체에 대한 1회 학습 반복을 epoch 이라 하며, 정확히 떨어지지 않을 경우 마지막 반복(+1)을 추가했다.

오차역전파법을 사용한 학습 구현 : Code

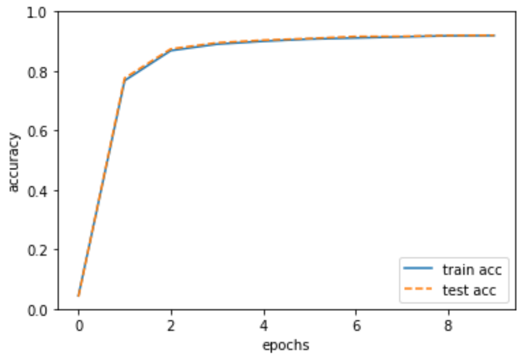
```python
# train
start_time = time.time()
for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    grad = network.gradient(x_batch, t_batch) # backpropagation

    for key in ('W1', 'b1', 'W2', 'b1'):
        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 or i == iters_num-1:
        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('%d iter train acc, test acc : %.4f, %.4f'
              % (i, train_acc, test_acc))
        print('%d iteration time = %.2f sec' % (i, time.time() - start_time))
```

```
0 iter train acc, test acc : 0.0437, 0.0411
0 iteration time = 0.48 sec
120 iter train acc, test acc : 0.7676, 0.7760
120 iteration time = 2.00 sec
240 iter train acc, test acc : 0.8686, 0.8741
240 iteration time = 3.50 sec
360 iter train acc, test acc : 0.8902, 0.8949
360 iteration time = 5.01 sec
480 iter train acc, test acc : 0.9000, 0.9036
480 iteration time = 6.48 sec
600 iter train acc, test acc : 0.9072, 0.9093
600 iteration time = 7.97 sec
720 iter train acc, test acc : 0.9110, 0.9161
720 iteration time = 9.45 sec
840 iter train acc, test acc : 0.9148, 0.9163
840 iteration time = 10.94 sec
960 iter train acc, test acc : 0.9183, 0.9196
960 iteration time = 12.43 sec
999 iter train acc, test acc : 0.9191, 0.9200
999 iteration time = 13.22 sec
```

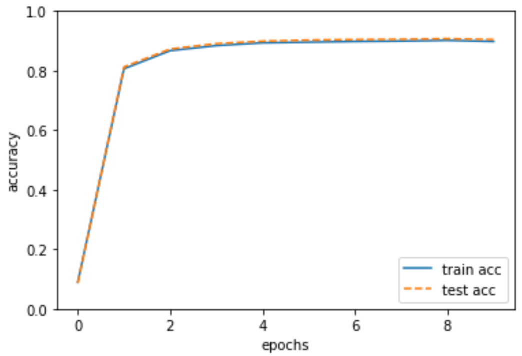
- $batch\_mask$: $0 \sim train_size - 1$까지의 정수 중에, $batch\_size$ 개수의 무작위 값을 갖는 numpy array
  ⇒ x_train과 x_batch에 적용하여 x_batch와 t_batch를 생성
- $network.gradient$를 통해 기울기 벡터($grad$)를 계산 후, 각 매개변수들에 대해 $learning\_rate$ 만큼의 경사하강법을 진행한다.
  ⇒ $gradient$(오차역전법)과 $numerical\_gradient$(수치 미분)의 차이 비교
- x_batch와 t_batch를 $net.loss$ 함수로 넘겨주면, $loss$를 return
  ⇒ $network.loss$ 내부에선 x_batch의 predict 결과와 t_batch 사이의 cross entropy error를 계산하여 $loss$로 return한다.
- epoch 마다 $network.accuracy$를 통해 훈련용 데이터와 테스트 데이터를 추정한 결과의 정확도, 걸린 시간을 print 해주며, list에 저장
  ⇒ 수치 미분을 이용했을 때는 같은 조건에서 40시간이 넘게 걸린 것에 비해, 오차역전파법을 이용했을 때는 13초 정도 밖에 걸리지 않았다!

오차역전파법을 사용한 학습 구현 : Code

```
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()
```



오차역전법을
적용한 학습
정확도 그래프



수치 미분을
적용한 학습
정확도 그래프

- *train_acc_list*를 plot한 그래프 위에 *test_acc_list*의 선 스타일이 다르도록 plot 해주고, 축 명과 y축 범위, 범례를 설정한다.
- 매 epoch에 따라 훈련용 데이터를 이용한 추정 결과 (*train_acc_list*) 값과 테스트 데이터를 이용한 추정 결과(*test_acc_list*) 값이 높아짐을 그래프로 알 수 있으며, 학습이 진행 되고 있음을 알 수 있다.
- *train_acc_list* 값과 *test_acc_list* 값이 거의 동일한 양상을 보이므로, 훈련용 데이터 과적합(overfitting)이 일어나지 않았음을 확인할 수 있다.
- 오차역전법을 적용한 학습 정확도 그래프와 수치 미분을 적용한 학습 정확도 그래프를 보면, 수치적으로는 차이가 없다고 판단할 수 있을 정도의 신뢰성을 보였으며 시간적으로 굉장한 이득을 보았다.