sigmoid関数

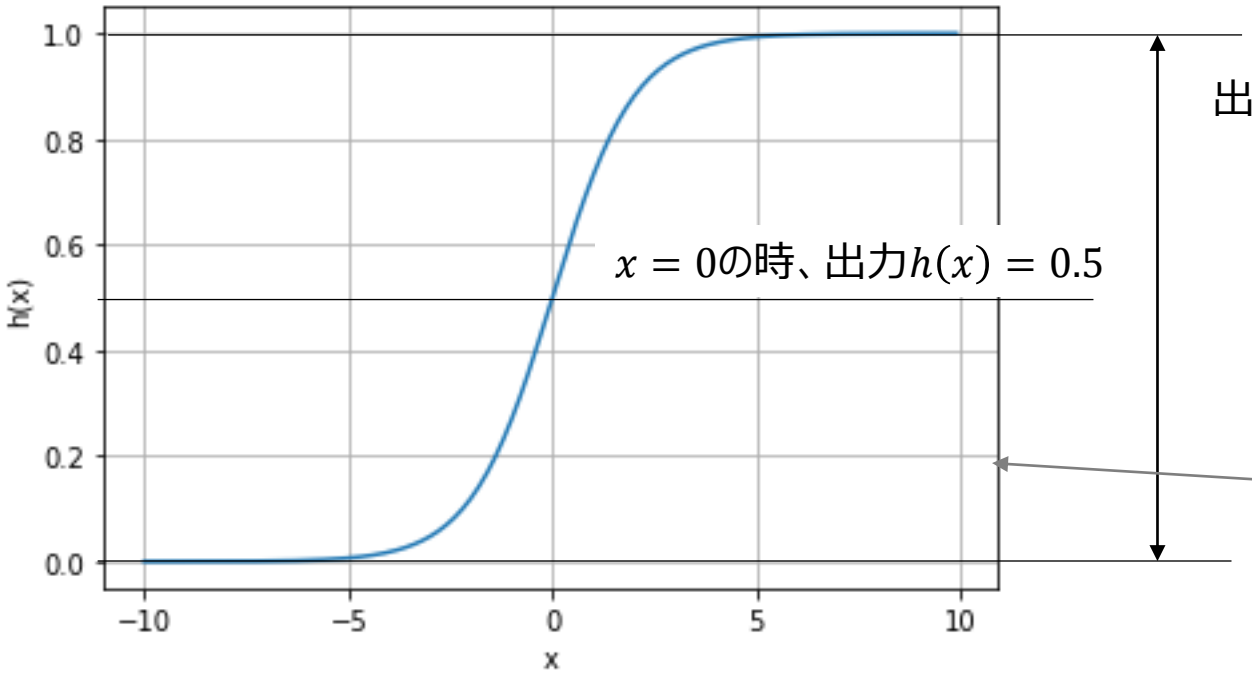
情報ソース：ゼロから作るDeep Learning

【定義】
・活性化関数の一つで非線形関数。

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

【実装例】

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```



$x = 0$の時、出力$h(x) = 0.5$

出力$h(x)$は0~1の間を取る

プロットする関数

```
def plot_function(x, function):

    y = function(x)
    plt.plot(x, y)
    plt.xticks([-10,-5,0,5,10])
    plt.grid()
    plt.xlabel("x")
    plt.ylabel("h(x)")
    plt.show()
```

ソフトマックス関数

情報ソース：ゼロから作るDeep Learning

【定義】
・出力系に用いられる関数で分類問題で使われる。
　(対し回帰問題では入力をそのまま出力する恒等関数が用いられる。)
・各出力は0~1.0の間の実数になる。また出力の総和は1になる。
・これを用いて複数のクラス分類でも、それぞれのクラスに属する確率(ただしマルチクラス問題。マルチラベル問題は不可)として扱う事ができる。

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_k)}$$

・実装上は、オーバーフローに対処するために任意の定数を用いて以下のようにする。

$$y_k = \frac{\exp(a_k + C')}{\sum_{i=1}^{n} \exp(a_k + C')}$$

・C'はどんな値でもよいが一般的には入力信号の最大の値を用いて、その値を引く。

【実装例】

```
def softmax(a): # aは出力ベクトル。
    c = np.max(a)
    exp_a = np.exp(a – c) # オーバーフロー対策
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a
    return y
```

ReLU関数

情報ソース：ゼロから作るDeep Learning

【定義】

・活性化関数の一つで非線形関数。

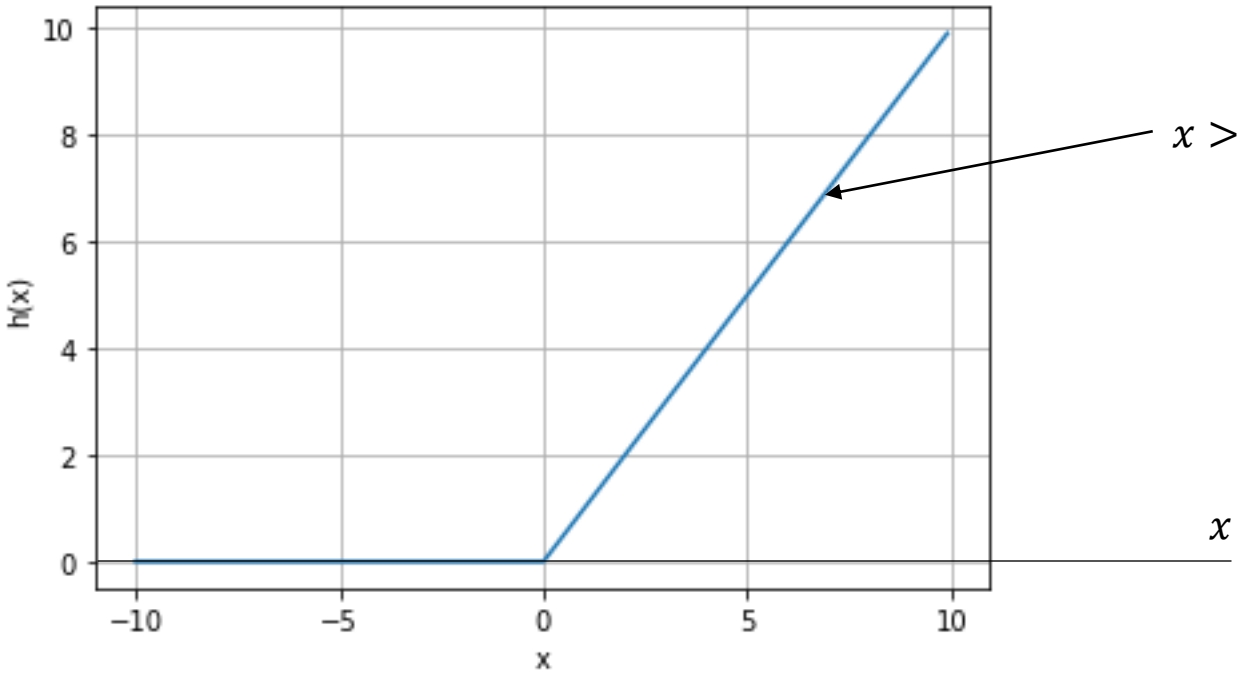
$$h(x) = \begin{cases} x\ (x > 0) \\ 0\ (x \leq 0) \end{cases}$$

・ReLU関数はx=0で微分できない。
そのため便宜的にx=0の時の微分値を1とする。

$$\frac{d}{dx}h(x) = \begin{cases} 1\ (x \geqq 0) \\ 0\ (x < 0) \end{cases}$$

【実装例】

```
def relu(x):
    return np.maximum(0, x)
```



$x > 0$の時、出力$h(x) = x$

$x \leqq 0$の時、出力$h(x) = 0$

Leaky ReLU関数

情報ソース：Neural Network Console 取り扱い説明書

【定義】
・活性化関数の一つで非線形関数。

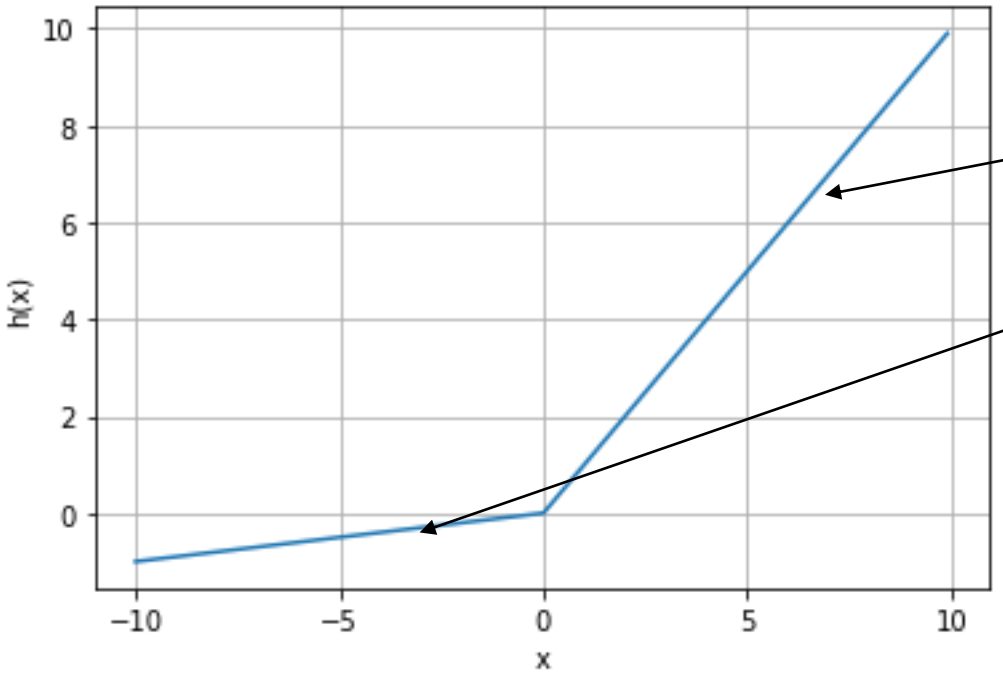
$$h(x) = \begin{cases} x\ (x > 0) \\ ax\ (x \leq 0) \end{cases}$$

・Leaky ReLU関数もReLU関数同様にx=0で微分できない。
そのため便宜的にx=0の時の微分値を1とする。

$$\frac{d}{dx}h(x) = \begin{cases} 1\ (x \geqq 0) \\ a\ (x < 0) \end{cases}$$

【実装例】

```
def leaky_relu(x, alpha):
    return np.where(x > 0.0, np.maximum(0, x), alpha * np.minimum(0, x))
```



$x > 0$の時、出力$h(x) = x$

$x \leqq 0$の時、出力$h(x) = ax$
*図のaはa=0.1

その他ReLU関数
【PReLU(Parametric ReLU)関数
・Leaky ReLUの定義のaの値を学習により獲得する方法。
【ReLU6】
・ReLU関数の出力の最大値を6とした関数。MobileNetで使用。

tanh関数

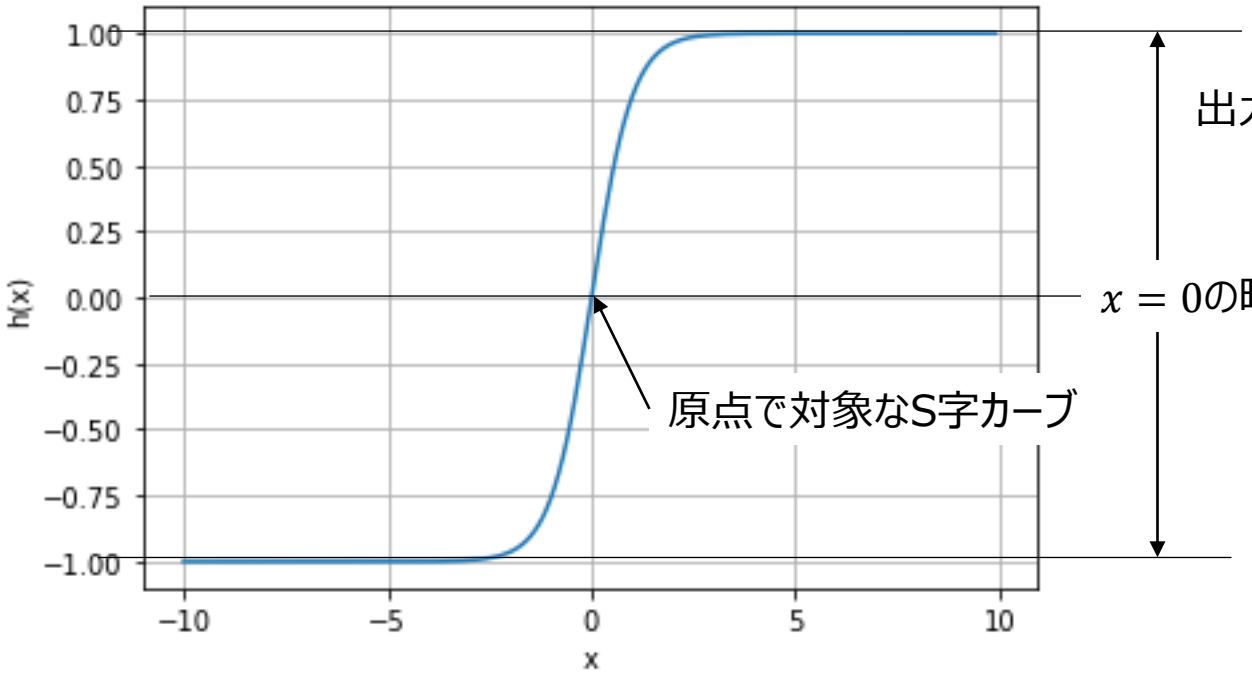
情報ソース：ゼロから作るDeep Learning2

【定義】
・活性化関数の一つで非線形関数。双曲線正弦関数やハイパボリック・タンジェント関数と呼ばれる。
・sigmoid関数の微分の最大値が0.25であるという弱点を解決するために提案されたもの。

$$h(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

【実装例】

```
def tanh(x):
    return (np.exp(x)-np.exp(-x)) / (np.exp(x)+np.exp(-x))
```



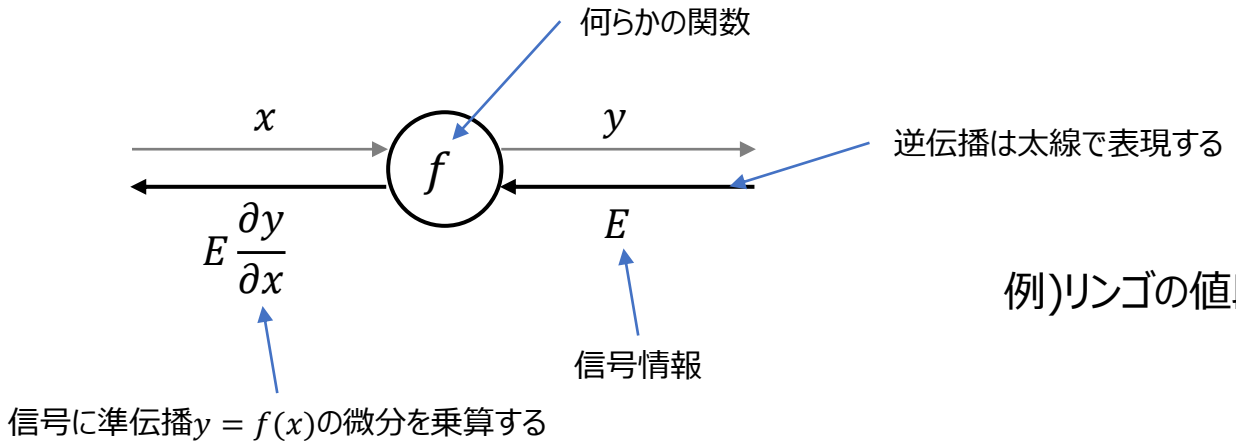
出力$h(x)$は-1～1の間を取る

$x = 0$の時、出力$h(x) = 0.0$

微積分の連鎖率

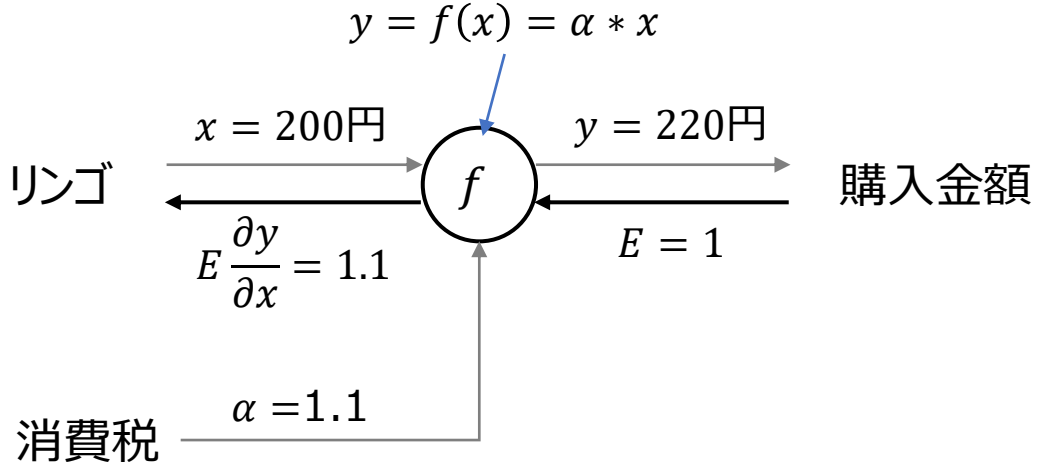
情報ソース：ゼロから作るDeep Learning

【計算グラフの逆伝播】
・計算グラフ上の逆伝播は微分を表す。
・以下のように記すことができる。



これは、xの値がyの値に加える影響の数値。

例)リンゴの値段



もしリンゴの値段が20円上がると、その影響は1.1倍、つまり2.2円影響するという事が、この計算グラフより読み取る事ができる。

微積分の連鎖率

情報ソース：ゼロから作るDeep Learning

【連鎖率】
・ある関数が合成関数で表される場合、その合成関数の微分は、合成関数を構成するそれぞれの関数の微分の積によって
あらわす事ができる。

例えば、$z = g(t),\ t = f(x, y)$の時、zへのｘの影響を微分で考える場合、

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

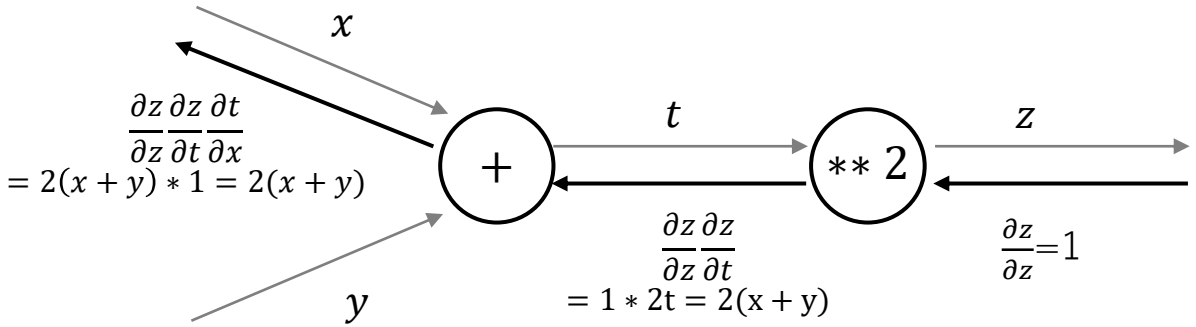
これは、∂tが互いに打ち消しあうことから分かる。

例) $z = t^2,\ t = x + y$の場合

$\frac{\partial z}{\partial t} = 2t,\ \frac{\partial t}{\partial x} = 1$により

$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t * 1 = 2(x + y)$と計算できる。

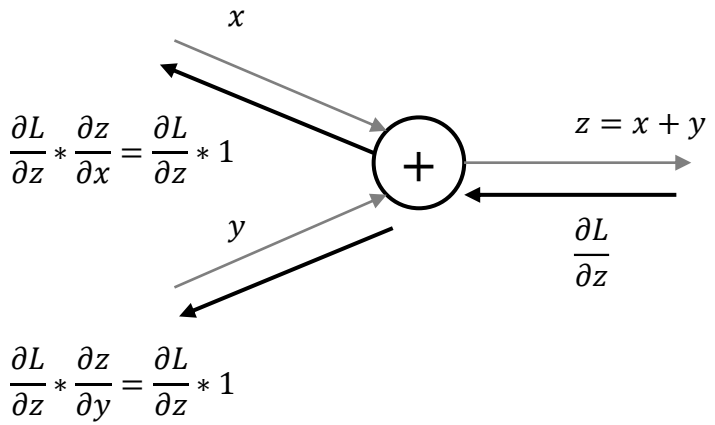
これを計算グラフで書くと



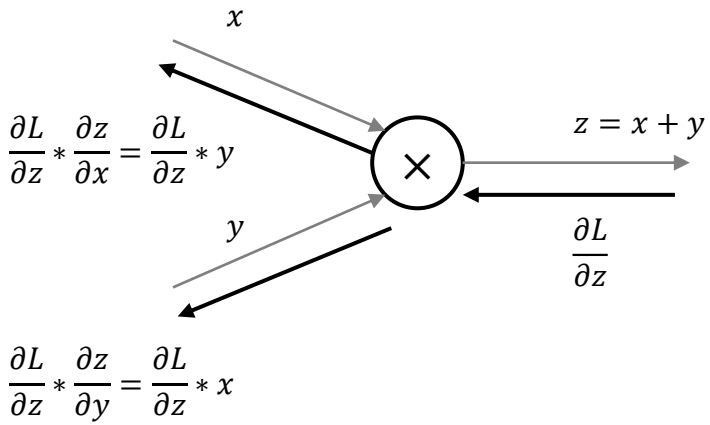
加算ノードと乗算ノードの順伝播と逆伝播

情報ソース：ゼロから作るDeep Learning

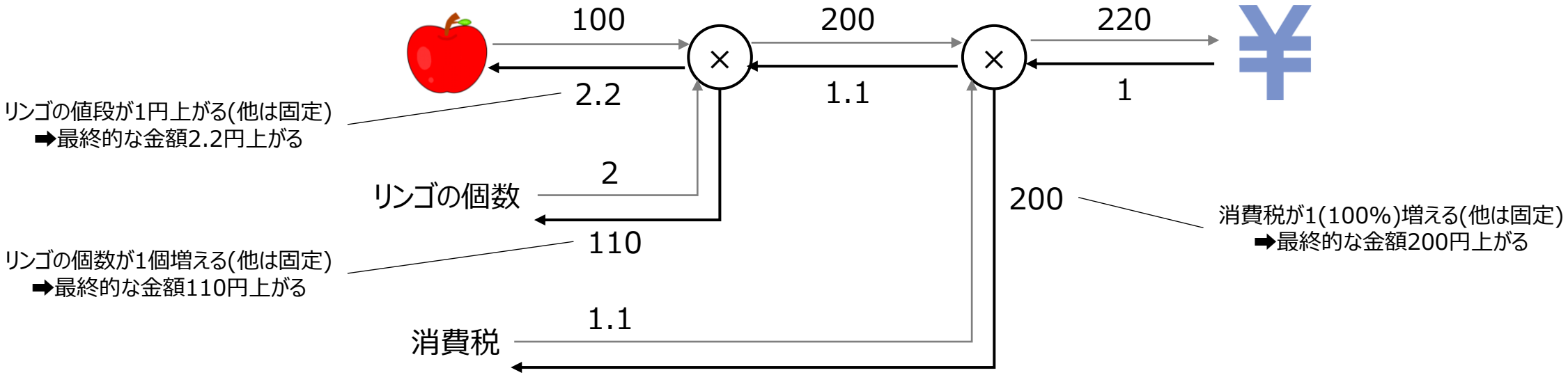
加算ノード



乗算ノード



例)



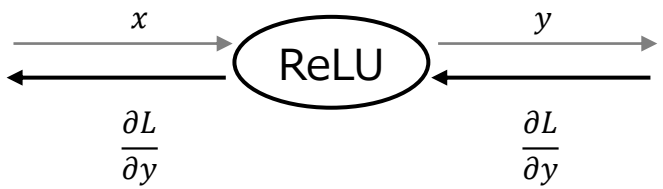
ReLUレイヤの順伝播と逆伝播

情報ソース：ゼロから作るDeep Learning

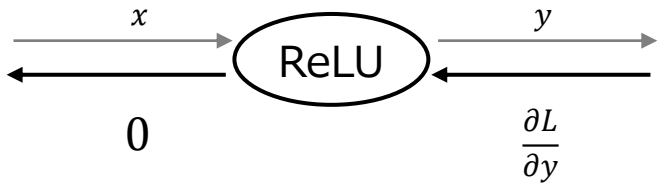
【ReLU関数の定義】

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

$x > 0$の時



$x \leq 0$の時



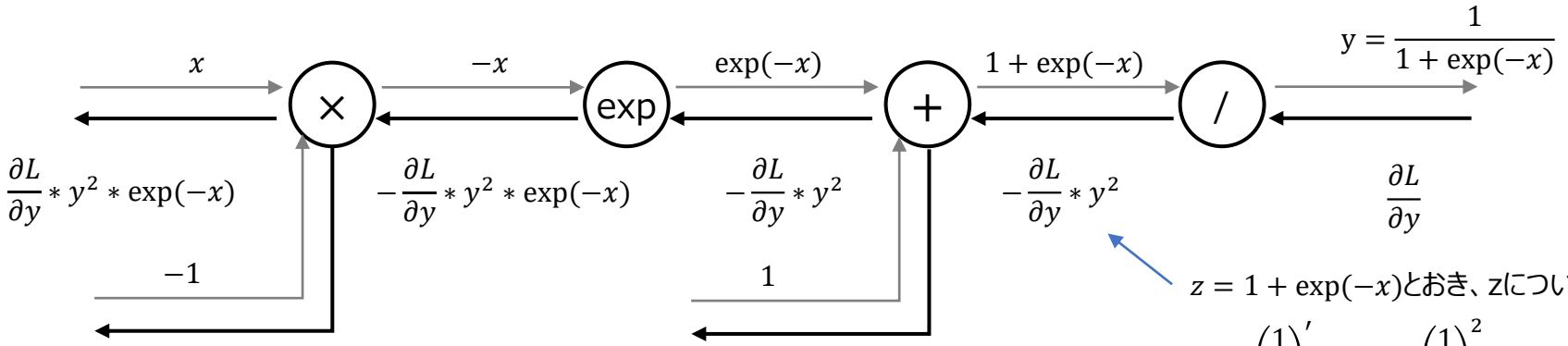

```
class Relu:

    def __init__(self):
        self.mask = None # 初期化

    def forward(self, x): # 順伝播
        self.mask = (x<=0) # xが0より小さい所を記憶。Trueをつける
        out = x.copy() # 元々の入力をコピー
        out[self.mask] = 0 # xが0より小さい所を0に変更する。そのほかはそのまま出力
        return out

    def backward(self, dout): # 逆伝播
        dout[self.mask] = 0 # 順伝播時に0だったところは、逆伝播を0に変換する。
        dx = dout #
        return dx
```

Sigmoidレイヤの順伝播と逆伝播

情報ソース：ゼロから作るDeep Learning

【Sigmoid関数の定義】

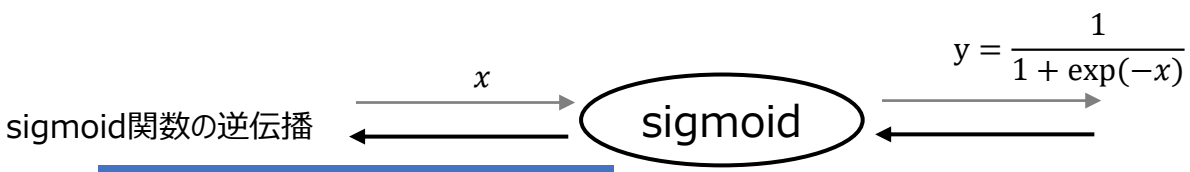
$$y = \frac{1}{1 + \exp(-x)}$$



$z = 1 + \exp(-x)$とおき、zについて微分すると

$$y' = \left(\frac{1}{z}\right)' = (-1)\left(\frac{1}{z}\right)^2 = (-1)\left(\frac{1}{1+\exp(-x)}\right)^2 = -y^2$$

まとめると、



$$\frac{\partial L}{\partial y} * y^2 * \exp(-x) = \frac{\partial L}{\partial y} * y(1-y)$$

$$y^2 * \exp(-x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \frac{\exp(-x)}{1+\exp(-x)} * \frac{1}{1+\exp(-x)} = \frac{-1+1+\exp(-x)}{1+\exp(-x)} * \frac{1}{1+\exp(-x)} = y(1-y)$$

```
class Sigmoid:

    def __init__(self):
        self.out = None # 初期化

    def forward(self, x): # 順伝播
        out = 1 / (1+np.exp(-x)) # 定義通り
        self.out = out # コピーしてインスタンスに記憶する。
        return self.out

    def backward(self, dout): # 逆伝播
        dx = dout * (1.0 – self.out) * self.out # 左図の計算式通り
        return dx
```

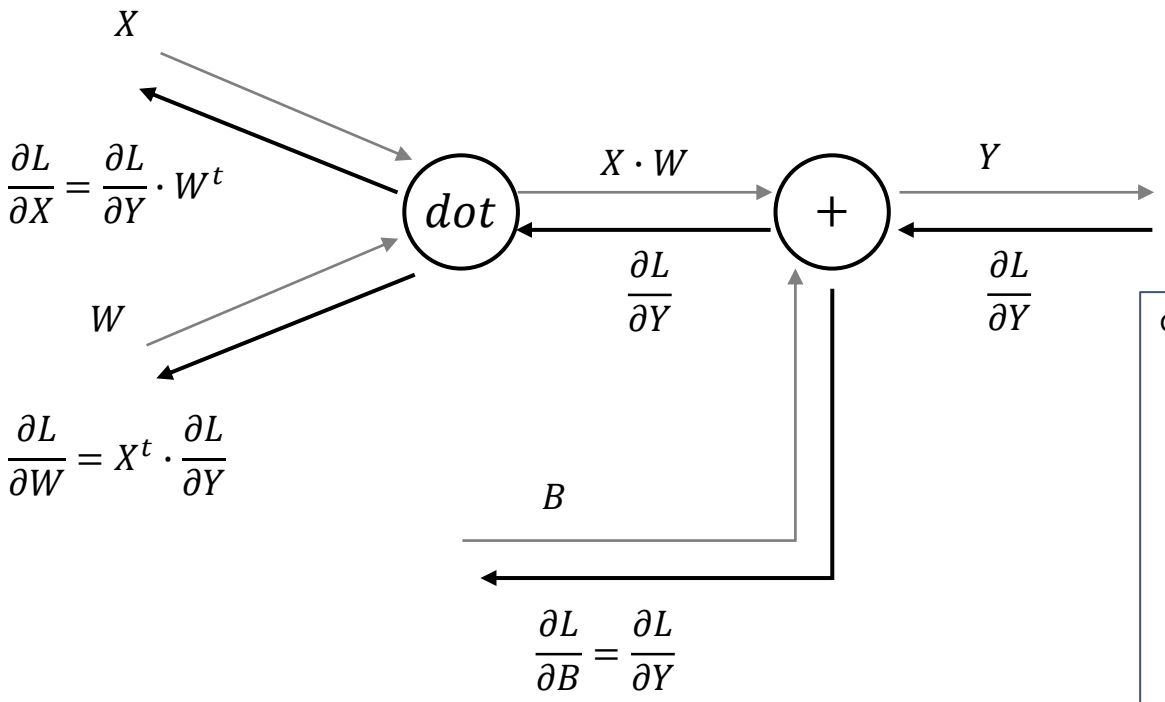
Affineレイヤの順伝播と逆伝播

情報ソース：ゼロから作るDeep Learning

【Affineレイヤの計算】

$Y = X \cdot W + B$

ここで、Y: 出力, X: 入力, W: 重み, B: バイアス



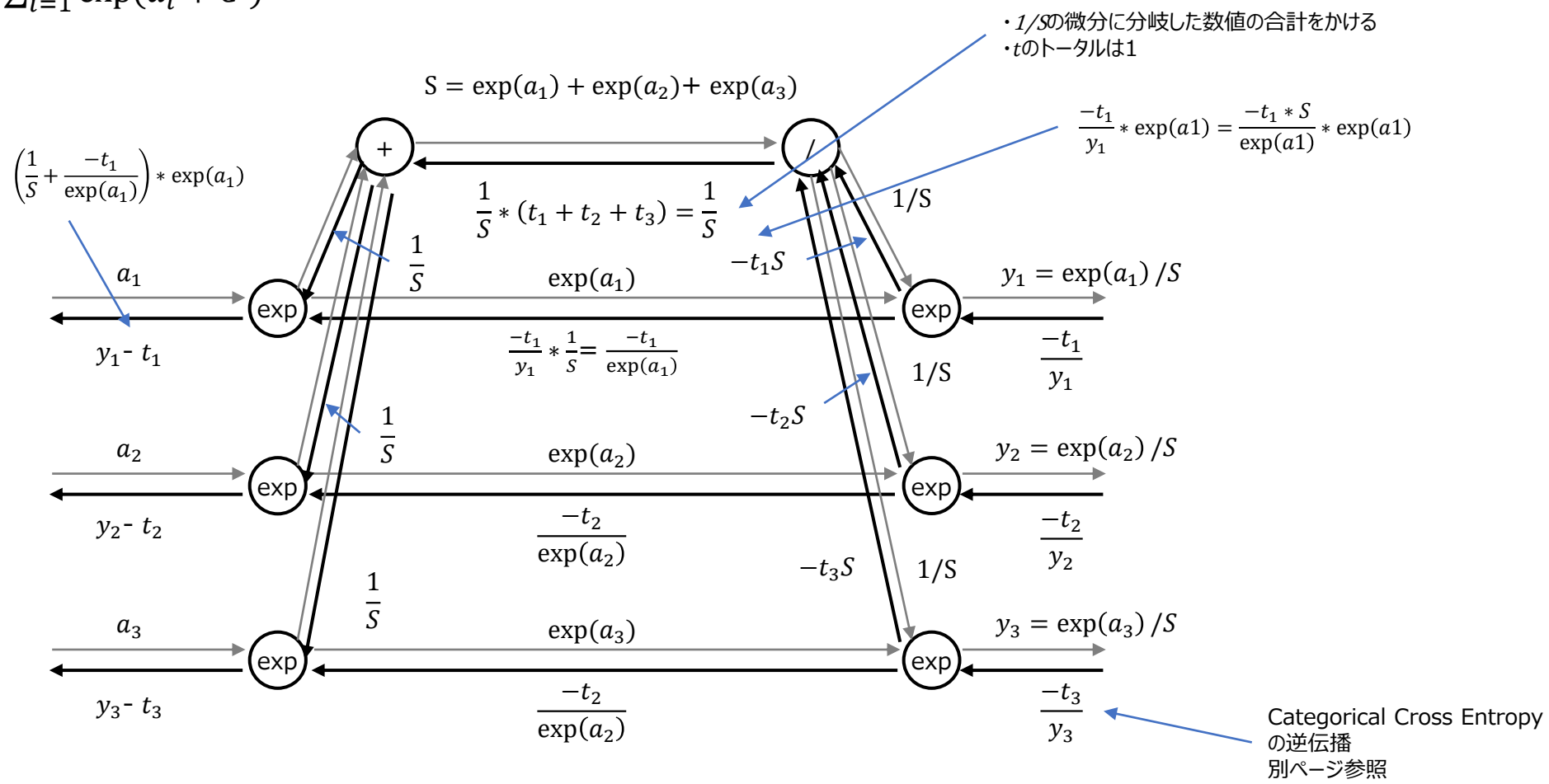

```
class Affine:

    def __init__(self, W, b): # 引数は初期値。通常ランダム。
        self.W = W # 重み
        self.b = b # バイアス
        self.x = None # 初期化
        self.dw = None # 初期化
        self.db = None # 初期化

    def forward(self, x): # 順伝播
        self.x = x # 入力を記憶
        out = np.dot(x, self.W) + self.b # 定義通りの計算
        return out

    def backward(self, dout): # 逆伝播
        dx = np.dot(dout, self.W.T) # 出力の微分値と重みの転置行列の積
        self.dW = np.dot(self.x.T, dout) # 入力の転置行列と出力の微分値の積
        self.db = np.sum(dout, axis=0) # バイアスは出力の微分値の総和で求める。
        return dx # dWとdBは学習する内部のパラメータなので出力しない。
```

Softmaxレイヤの順伝播と逆伝播

情報ソース：ゼロから作るDeep Learning

【Softmaxレイヤの計算】

$$y_k = \frac{\exp(a_k + C')}{\sum_{i=1}^{n} \exp(a_i + C')}$$

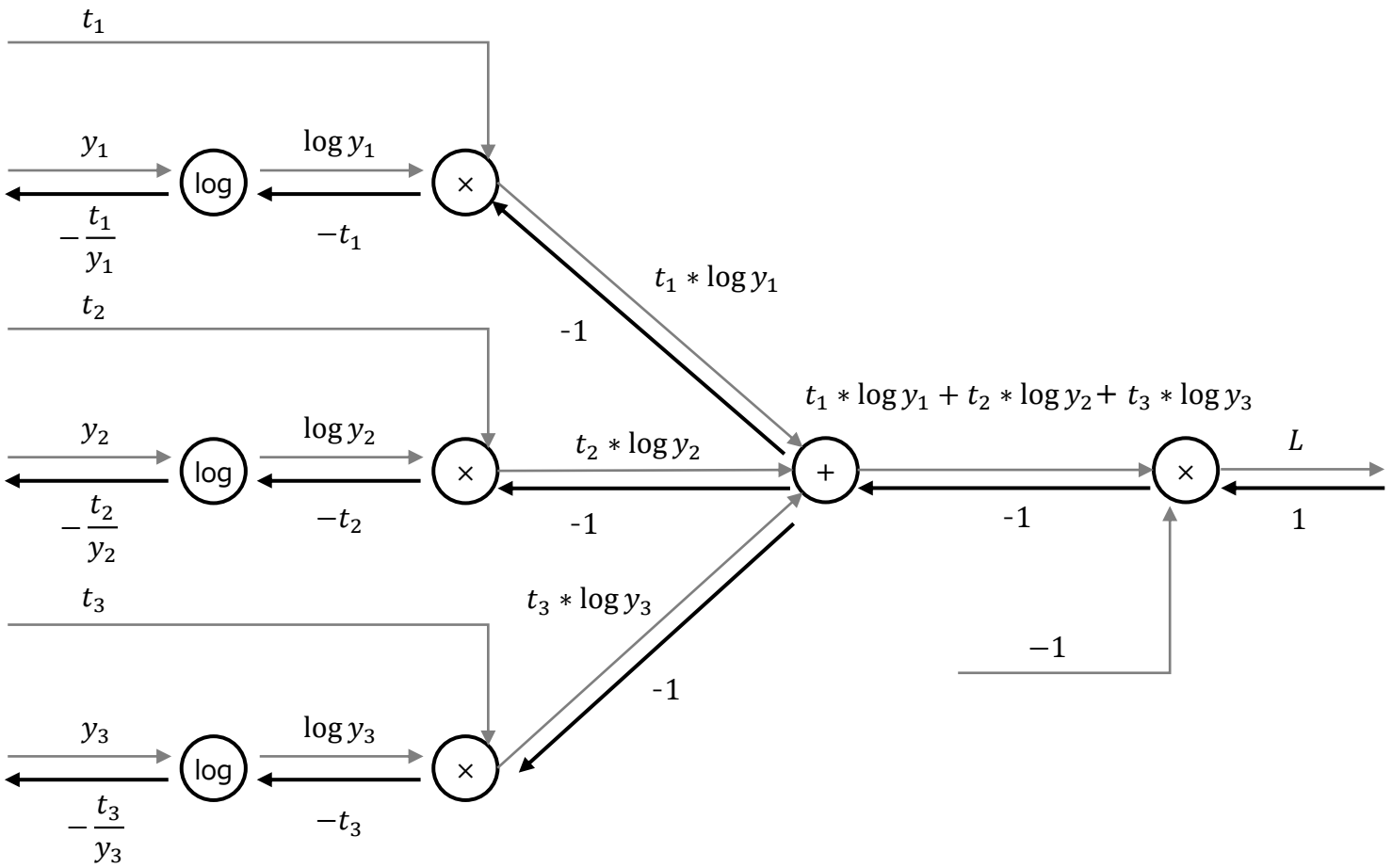


Cross Entropy Errorレイヤの順伝播と逆伝播

情報ソース：ゼロから作るDeep Learning

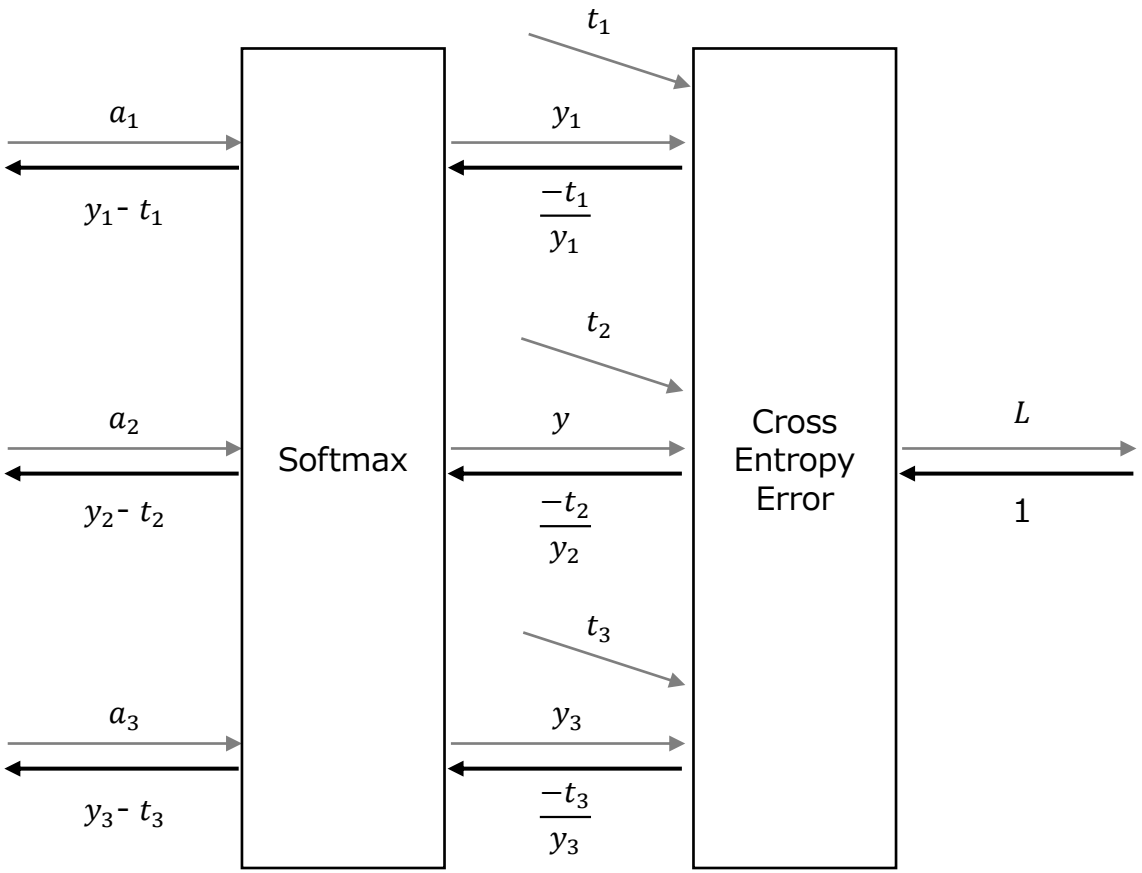
【Cross Entropy Errorレイヤの計算】

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



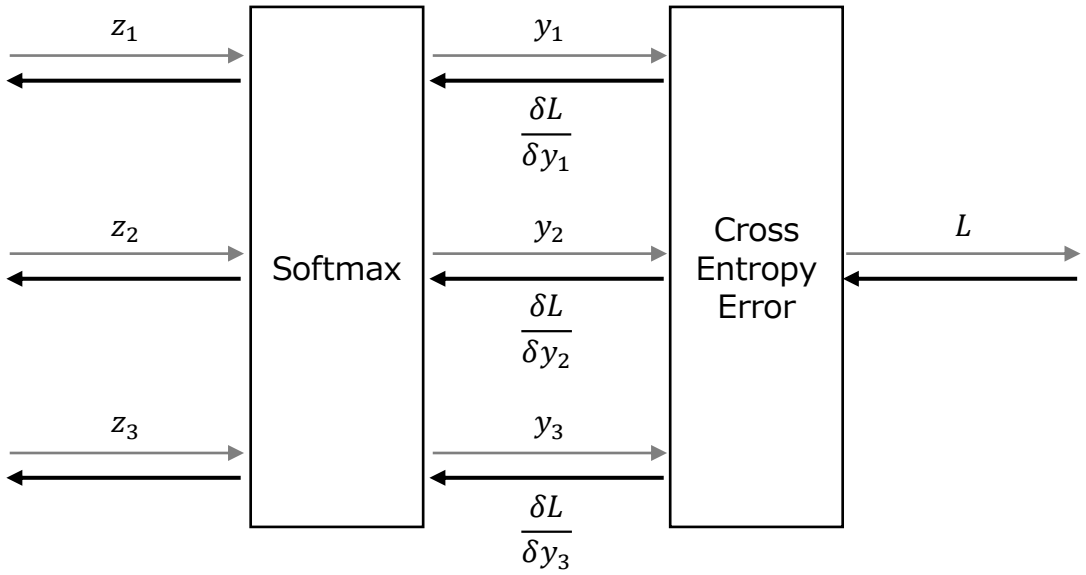
Softmax-with-Lossレイヤの順伝播と逆伝播

【Softmax-with-Lossレイヤ】
SoftmaxレイヤとCross Entropy Errorレイヤを組み合わせる、
簡易的に書くと以下のようになる。




```
class SoftmaxWithLoss:

    def __init__(self):
        self.loss = None # 損失
        self.y = None # softmaxの出力
        self.t = None # 教師データ

    def forward(self, x, t): # 順伝播
        self.t = t
        self.y = softmax(x) # ソフトマックス関数を参照
        self.loss = cross_entropy_error(self.y, self.t) # クロスエントロピー誤差関数
        return self.loss

    def backward(self, dout=1)
        batch_size = self.t.shape[0] # 加えられて教師データのサイズからバッチサイズ取得
        dx = (self.y – self.t) / batch_size
        return dx
```

*Softmaxレイヤの順伝播と逆伝播



【Softmaxレイヤ単独での実装計算】

1. $y_k = \frac{\exp(a_k + C\prime)}{\sum_{i=1}^{n} \exp(a_k + C\prime)}$

$\frac{\delta y_i}{\delta z_k} = y_i(\delta_{ik} - y_k)$, δ_{ik}:クロネッカーのデルタ(i=kの時1, i≠kの時0)

↑

↑

$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$

$i = k$の時

$$\frac{\delta y_i}{\delta z_i} = \frac{\exp(z_i)\sum\exp(z_i) - \exp(z_i)\exp(z_i)}{(\sum\exp(z_i))^2} = \frac{\exp(z_i)}{\sum\exp(z_i)} - \frac{(\exp(z_i))^2}{(\sum\exp(z_i))^2} = y_i - {y_i}^2$$

$i \neq k$の時

$$\frac{\delta y_i}{\delta z_k} = \frac{\delta}{\delta z_k}\left(\frac{\exp(z_i + C')}{\sum_{i=1}^{n}\exp(z_k + C')}\right) = \exp(z_i) * (-1) * \frac{\exp(z_k)}{(\sum\exp(z_i))^2} = -\frac{\exp(z_i)}{\sum\exp(z_i)} * \frac{\exp(z_k)}{\sum\exp(z_i)} = -y_i y_k$$

求めたいのは$\frac{\delta L}{\delta z_k}$, これを連鎖率により分解して、

$$\frac{\delta L}{\delta z_k} = \sum_i \frac{\delta L}{\delta y_i}\frac{\delta y_i}{\delta z_k} = \sum_i \frac{\delta L}{\delta y_i} y_i(\delta_{ik} - y_k)$$

$$= \sum \frac{\delta L}{\delta y_i} y_i \delta_{ik} - \sum \frac{\delta L}{\delta y_i} y_i y_k = \frac{\delta L}{\delta y_i} y_k - \sum_i \frac{\delta L}{\delta y_i} y_i y_k$$

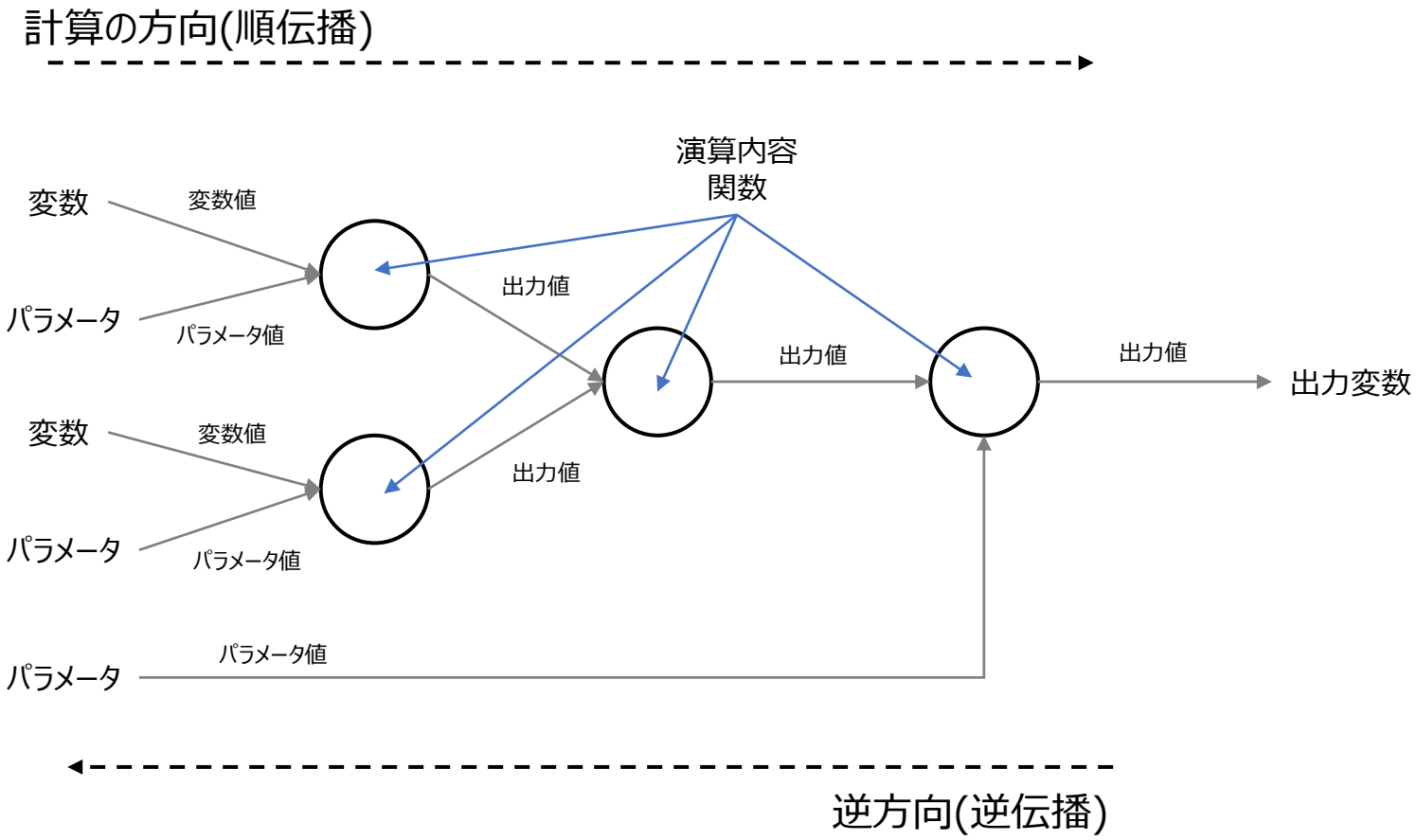
i=kだけが残るので

```
class Softmax:

    def __init__(self):
        self.params, self.grads = [], [] # パラメータと勾配の記憶用
        self.out = None # softmaxの出力

    def forward(self, x): # 順伝播
        self.out = softmax(x) # ソフトマックス関数を参照
        return self.out

    def backward(self, dout):
        dx = self.out*dout # 出力yと逆伝播のベクトルの積, δL/δy_i * y_kの部分でベクトルで計算される。
        sumdx = np.sum(dx, axis=1, keepdims=True) # Σ_i δL/δy_i * y_i の部分
        dx = -= self.out*sumdx # y_kをかけて元のdxから引く
        return dx
```

計算グラフ

情報ソース：ゼロから作るDeep Learning

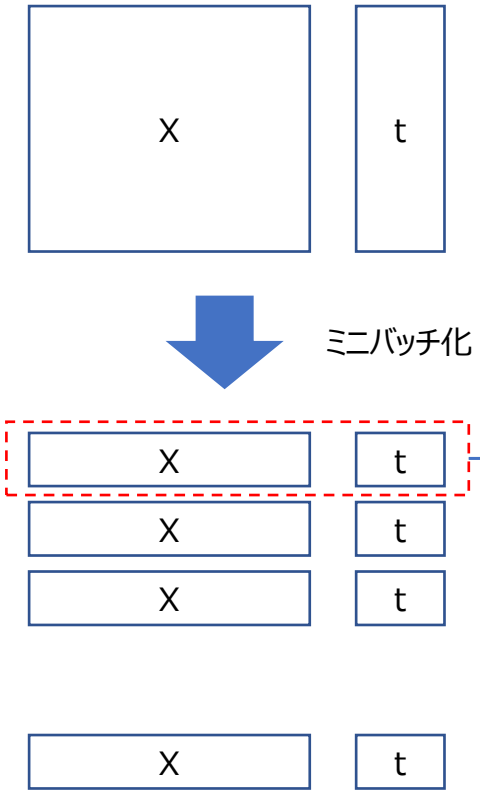
【計算グラフ】
・計算の過程をグラフで表したもの。
・グラフとはデータ構造としてのグラフであり、複数のノードとエッジによって表現される。
・局所的な計算を伝播することによって最終的な結果を得る事ができる。



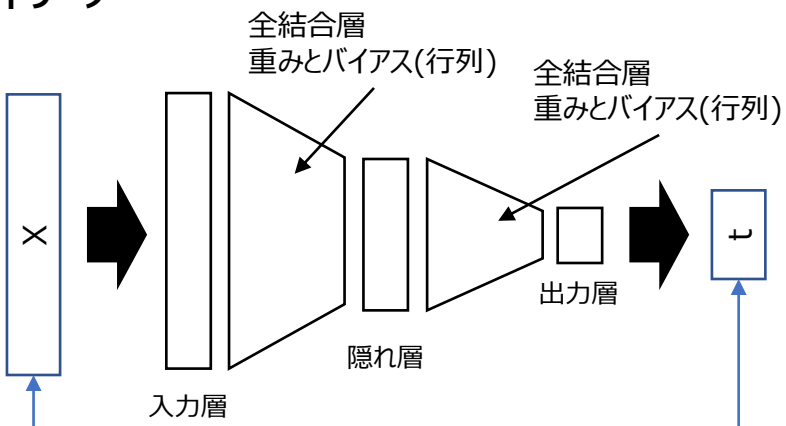
モデルの学習の実装

情報ソース：ゼロから作るDeep Learning

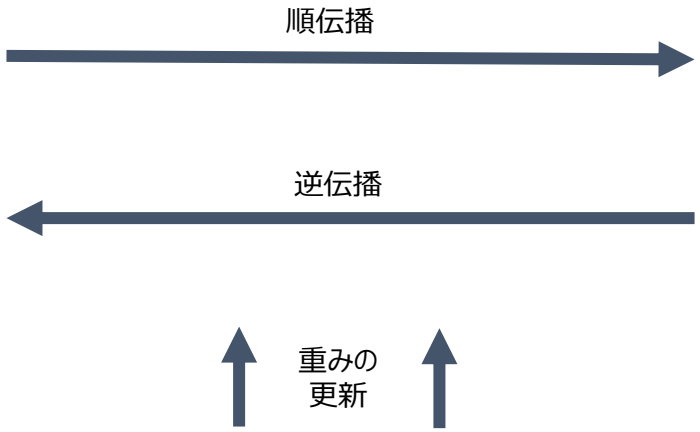
データセット



ネットワーク



学習のさせ方



(1)順伝播させて予測値yを求める。
予測値yと目標値tから目的関数を計算する。
その時に入力方向の情報を保持する。

(2)目的関数の値を起点に、逆伝播させる。
(1)で保持していた入力情報と重みの情報に、逆伝播を後ろから伝えていき、重みの勾配を計算し、保持する。

(3)保持していた重みと重みの勾配、学習率を使って重みを更新する。

(1)~(3)を繰り返し実行する。