5章　誤差伝播法

5.1 計算グラフ

❖これまで

- ニューラルネットワークの学習…ニューラルネットワークの重みパラメータの損失関数の勾配によって最小値へと学習する

☹計算に時間がかかってしまう

❖ここから

- 重みパラメータの勾配の計算を効率良く行う手法の「誤差伝搬法」について学ぶ

正しく理解するために…

⑴ 数式によって理解

⑵**「計算グラフ」**によって理解　←本章ではこれ！視覚的に理解を深める。

❖計算グラフ：計算の過程をグラフ（データ構造のことでノードとエッジによって表現）によって表したもの

簡単な問題を解いていく

→最終的に誤差伝搬法に！

5.1.1 計算グラフで解く

❖「計算グラフ」を使って簡単な問題を解く
　計算グラフ…ノードとエッジ(矢印)によって計算の過程を表すもの。
　ノードの中に、演算の内容を書く。矢印の上には計算の途中結果を書く。

問１：太郎くんはスーパーで１個100円のりんごを２個買いました。支払う金額を求めなさい。ただし、消費税が10%適用されるものとします。

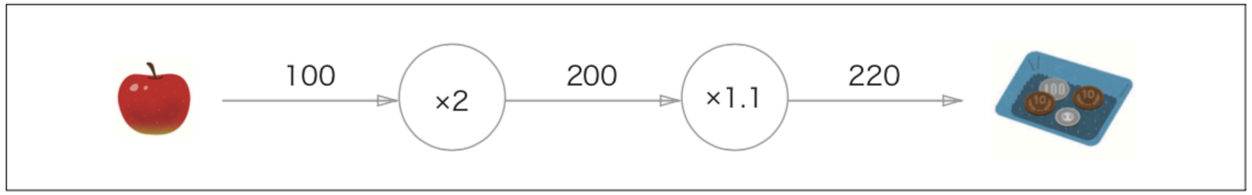


図5-1　計算グラフによる問 1 の答え

＜別の表し方＞

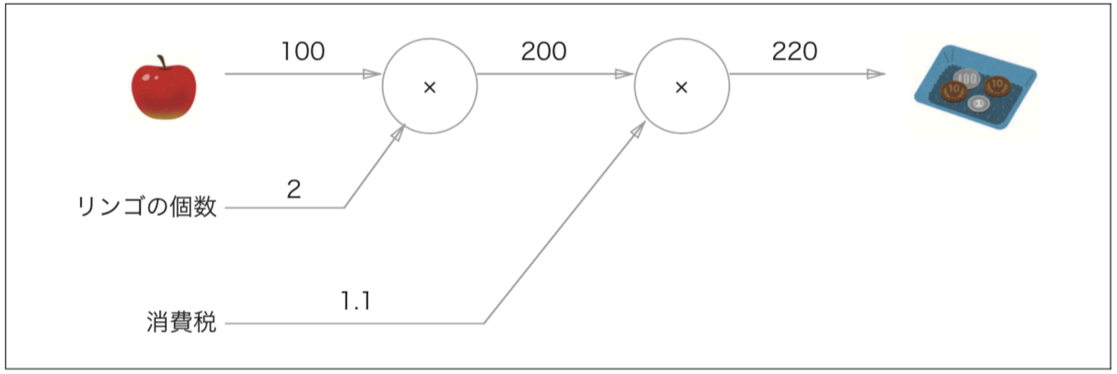


図5-2　計算グラフによる問 1 の答え：「リンゴの個数」と「消費税」を変数として、○の外に表記する

5.1.1 計算グラフで解く

問２：太郎くんはスーパーでりんごを２個、みかんを３個買いました。りんごは１個100円、みかんは１個150円です。消費税が10%かかるものとして、支払う金額を求めなさい。

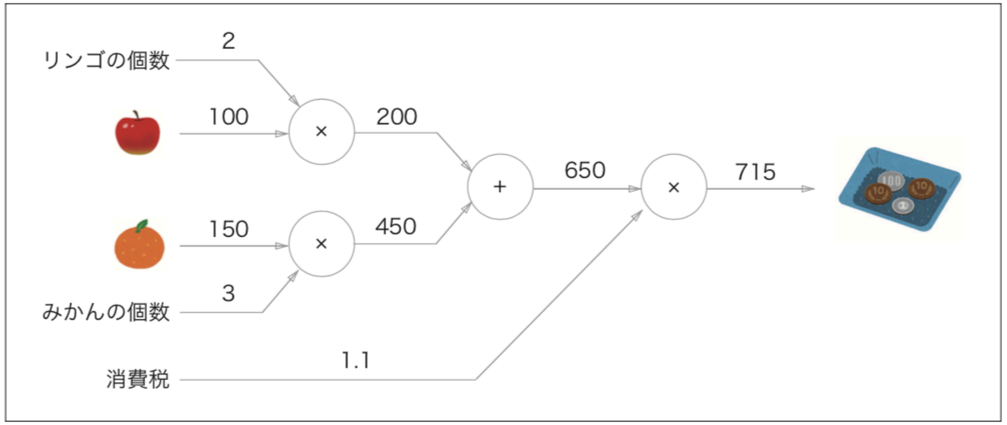


図5-3　計算グラフによる問２の答え

❖計算グラフを使って問題を解く。

1. 計算グラフを構築する
2. 計算グラフ上で計算を左から右へ進める(**順伝播**)

逆伝播…図でいう右から左の方向。この先の微分の計算で重要。

5.1.2 局所的な計算

❖計算グラフの特徴…「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ことができる

例：スーパーでりんごを２個と、それ以外にたくさんの買い物をする場合

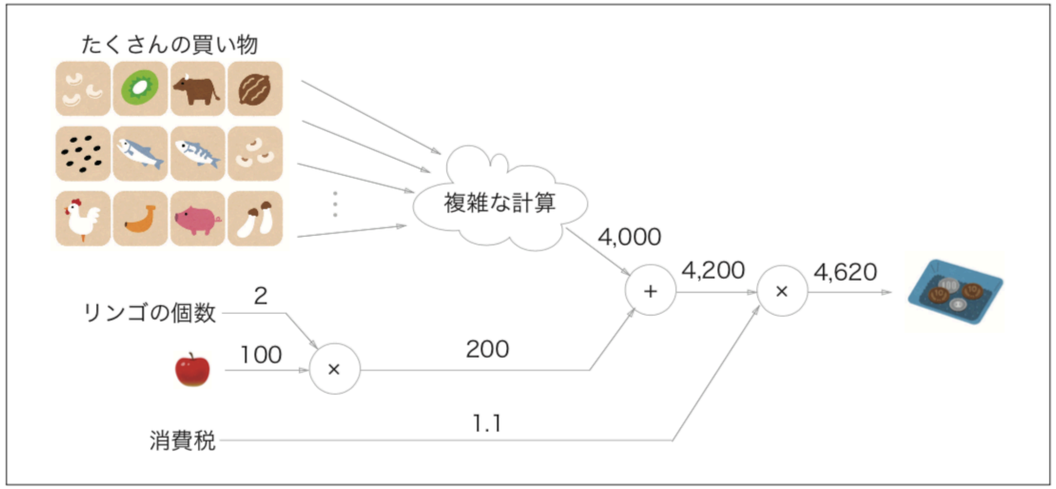


図5-4　リンゴ２個とそれ以外のたくさんの買い物の例

→局所的に計算を行うだけで、求まる。全体のことについては考えなくて良い。

5.1.3 なぜ計算グラフで解くのか？

❖計算グラフの利点

1. 「局所的な計算」- 全体がどんなに複雑な計算であっても各ノードでの単純な計算を行っていくことで問題を単純化できる
2. 途中の計算の結果を全て保持することができる点
3. **逆方向の伝播によって「微分」を効率良く計算できる点**

❖計算グラフの逆伝播について、問１の問題をもう一度考える。

問１：太郎くんはスーパーで１個100円のりんごを２個買いました。支払う金額を求めなさい。ただし、消費税が10%適用されるものとします。

例：りんごの値段が値上がりした場合、最終的な支払い金額にどのように影響するか

→「りんごの値段に関する支払い金額の微分」を求める

りんごの値段をx、支払い金額をLとすると、$\frac{\partial L}{\partial x}$を求めるということ。（りんごの値段が少し値上がりした時に支払い金額がどれだけ増加するか）

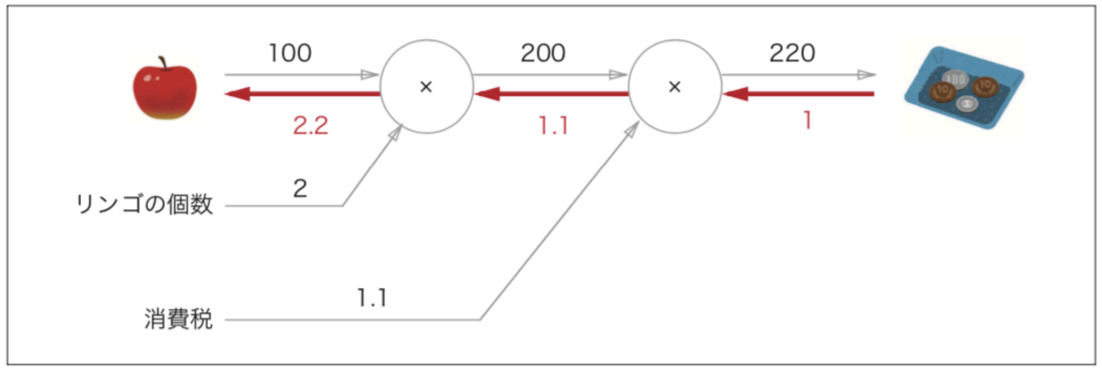


図5-5　逆伝播による微分値の伝達

→りんごが１円値上がりしたら

最終的な値段が2.2円増えるということ

5.2 連鎖律
5.2.1 計算グラフの逆伝播

❖「局所的な微分」を伝達する原理…**連鎖律**によるもの

計算グラフを使った逆伝播の例： $y = f(x)$

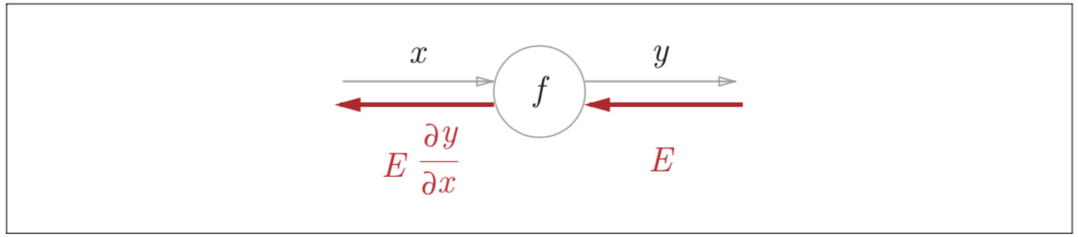


図5-6　計算グラフの逆伝播：順方向とは逆向きに、局所的な微分を乗算する

❖逆伝播のポイント：目的とする微分の値を効率良く求めることができる→連鎖律の説明へ

5.2.2 連鎖律とは

❖**合成関数**…複数の関数によって構成される関数のこと

❖**連鎖律の原理**＝合成関数の微分についての性質

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表すことができる

例：$z = (x+y)^2$

$$
\begin{aligned}
z &= t^2 \\
t &= x+y
\end{aligned}
$$

この時の$\frac{\partial z}{\partial x}$（xに関するzの微分）を求めると次のように表すことができる

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

それぞれ求める

$$
\begin{aligned}
\frac{\partial z}{\partial t} &= 2t \\
\frac{\partial t}{\partial x} &= 1
\end{aligned}
$$

それぞれの積を計算する

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

5.2.3 連鎖律と計算グラフ

❖連鎖律の計算を計算グラフで表してみる

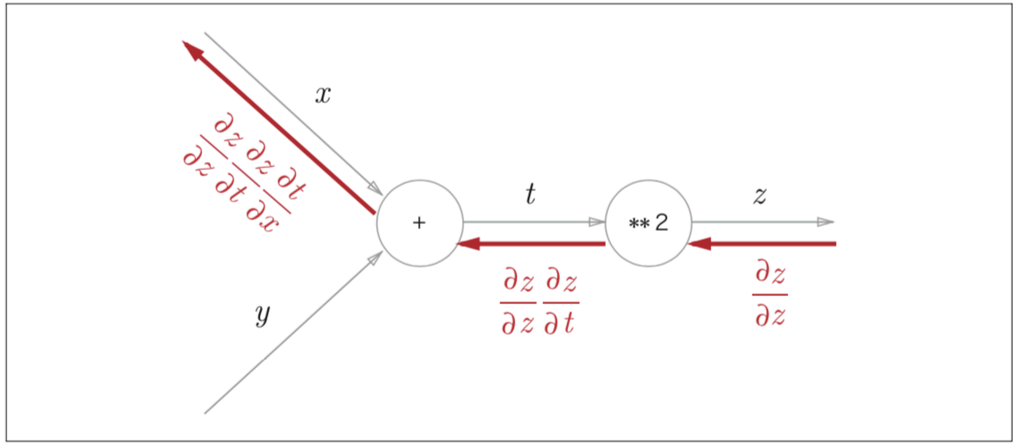


図5-7　式 (5.4) の計算グラフ：順方向とは逆向きの方向に、局所的な微分を乗算して渡していく

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

xに関するzの微分

先ほどの例を当てはめてみる

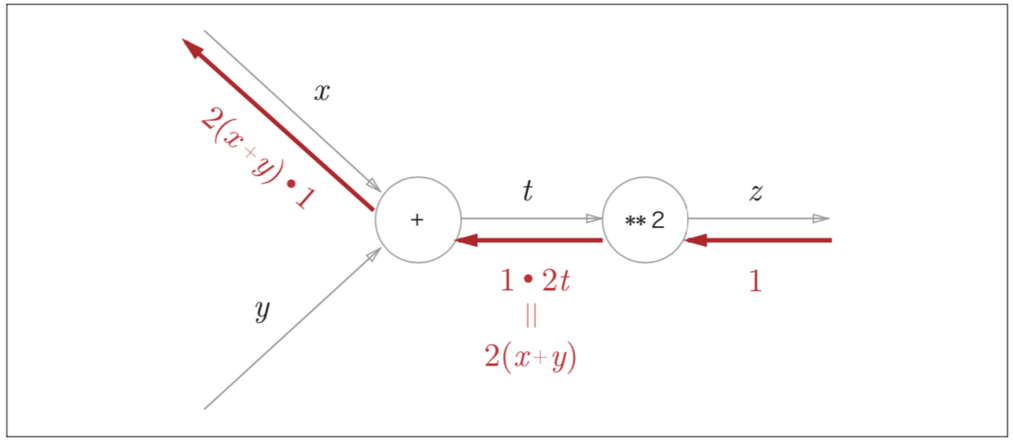


図5-8　計算グラフの逆伝播の結果より、$\frac{\partial z}{\partial x}$ は $2(x+y)$ となる

5.3 逆伝播
5.3.1 加算ノードの逆伝播

❖加算ノードの逆伝播

例：$z = x + y$

zの微分について考える

$$\frac{\partial z}{\partial x} = 1$$
$$\frac{\partial z}{\partial y} = 1$$

これを計算グラフに表す、この例では上流から伝わった微分値を$\frac{\partial L}{\partial z}$とした

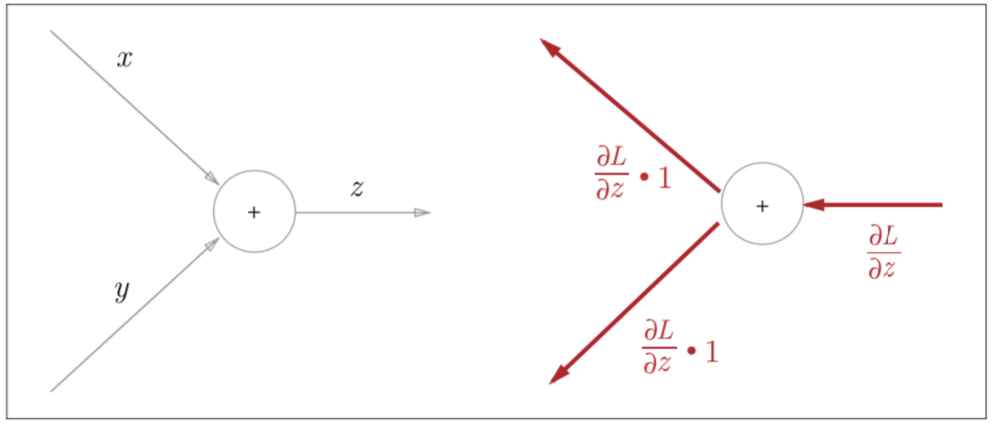


図5-9　加算ノードの逆伝播：左図が順伝播、右図が逆伝播。右図の逆伝播が示すように、加算ノードの逆伝播は、上流の値をそのまま下流へ流す

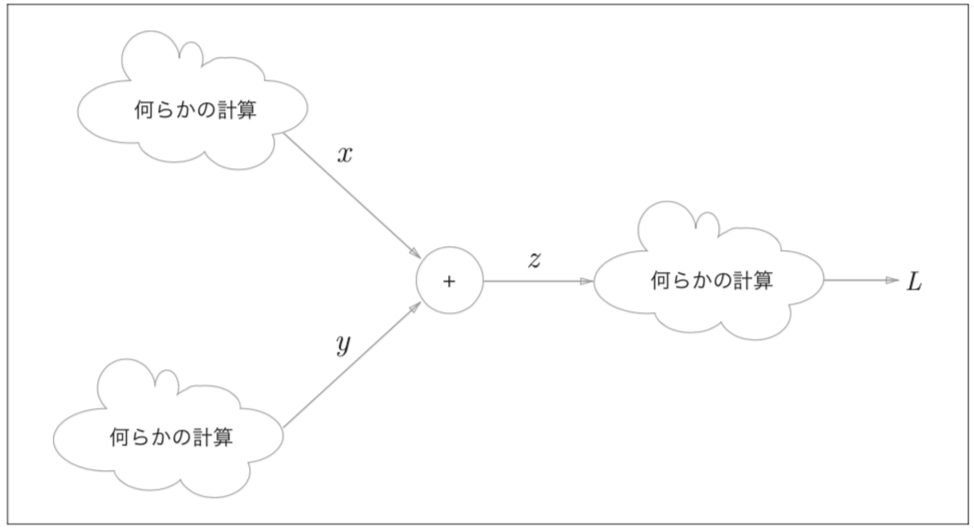


図5-10　最終的に出力する計算の一部に、今回の加算ノードが存在する。逆伝播の際には、一番右の出力からスタートして、局所的な微分がノードからノードへと逆方向に伝播されていく

5.3.1 加算ノードの逆伝播

具体例：10+5=15の計算、上流から1.3の値が流れてくるとする。

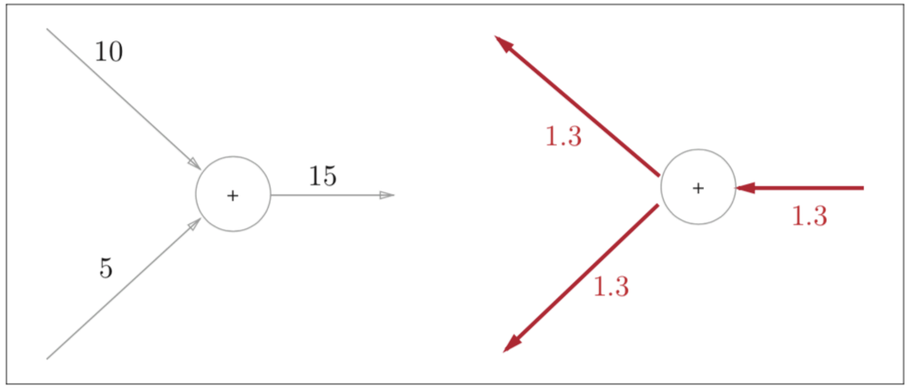


図5-11　加算ノードの逆伝播の具体例

❖加算ノードの逆伝播…入力信号を次のノードへ出力するだけ

5.3.2 乗算ノードの逆伝播

例：$z = xy$

zの微分を計算する

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

計算グラフで表す

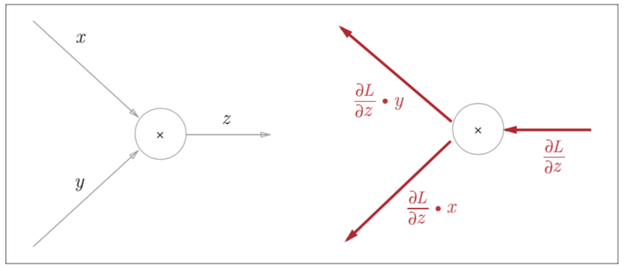


図5-12　乗算の逆伝播：左図が順伝播、右図が逆伝播

→”ひっくり返した値”を下流へ流す

・順伝播の際にxの信号であればy、yの信号であればxをという意味

具体例：10×5=50、上流から1.3の値が流れてくるとする

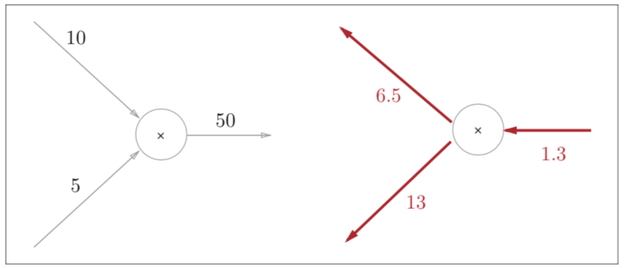


図5-13　乗算ノードの逆伝播の具体例

→加算ノードの場合：入力信号は必要なし

乗算ノードの場合：入力信号必要あり

乗算ノードの実装時には順伝播の入力信号を保持する

5.3.3 リンゴの例

例：リンゴ２個と消費税
解きたいこと…りんごの値段、りんごの個数、消費税の３つの変数が最終的な支払金額にどのように影響するかということ
=「りんごの値段に関する支払い金額の微分」、「りんごの個数に関する支払い金額の微分」、「消費税に関する支払い金額の微分」を求める

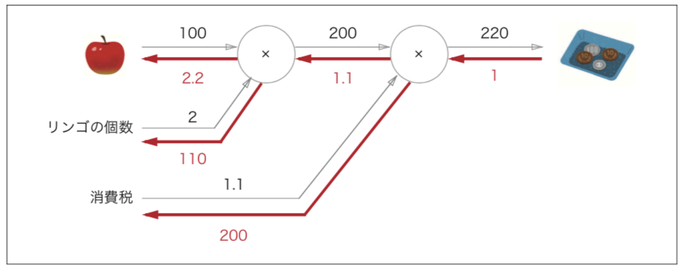


図5-14　リンゴの買い物の逆伝播の例

結果：りんごの値段の微分は2.2、りんごの個数の微分は110、消費税の微分は200。
　　消費税が1(100%)増加すると200、りんごの値段が1(1円)増加すると2.2の大きさが最終的に影響を与える

5.3.3 リンゴの例

例：りんごとみかんの買い物の逆伝播

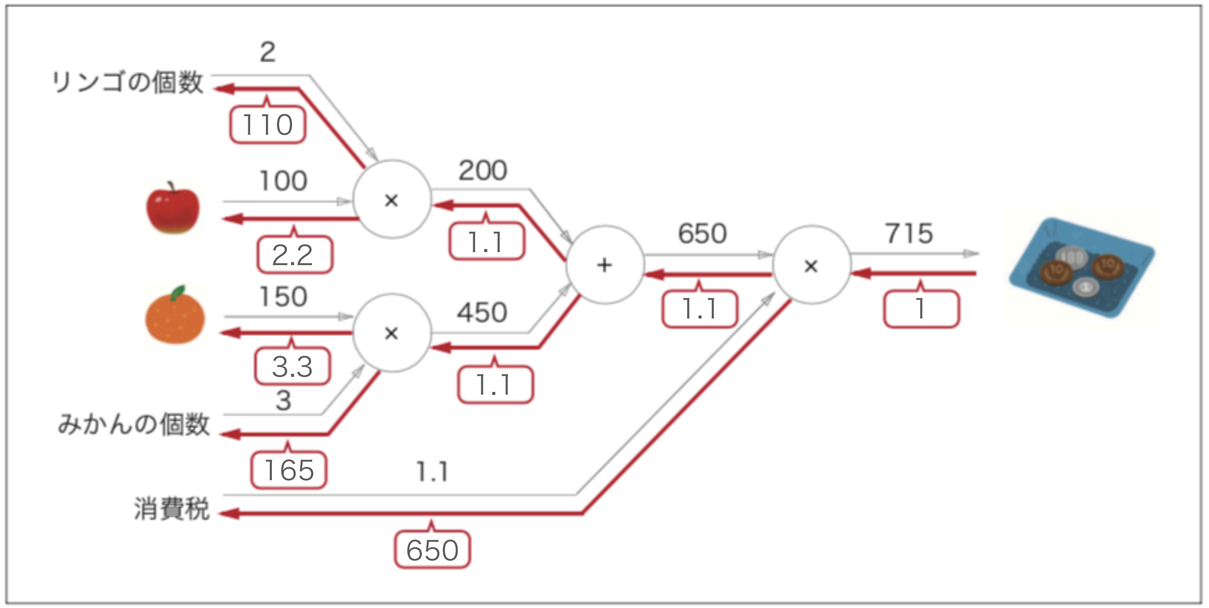


図 5-15　リンゴとみかんの買い物の逆伝播の例：四角に数字を入れて逆伝播を完成させよう

5.4 単純なレイヤの実装
5.4.1 乗算レイヤの実装

❖「リンゴの買い物」の例をPythonで実装していく。
計算グラフの乗算ノードを「乗算レイヤ(MulLayer)」、加算ノードを「加算レイヤ(AddLayer)」という名前で実装する
各レイヤはクラスで実装することにする

```python
# coding: utf-8


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

```

❖forward()_順伝播とbackward()_逆伝播という共通のメソッドを持つ
❖__init()__()：xとyの初期化＋順伝播時の入力値を保持する

・dout：上流から伝わってきた微分

ch05/layer_naive.py

5.4.1 乗算レイヤの実装

❖MulLayerを使って「りんごの買い物」を実装する

```python
# coding: utf-8
from 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)

```

ch05/buy_apple.py

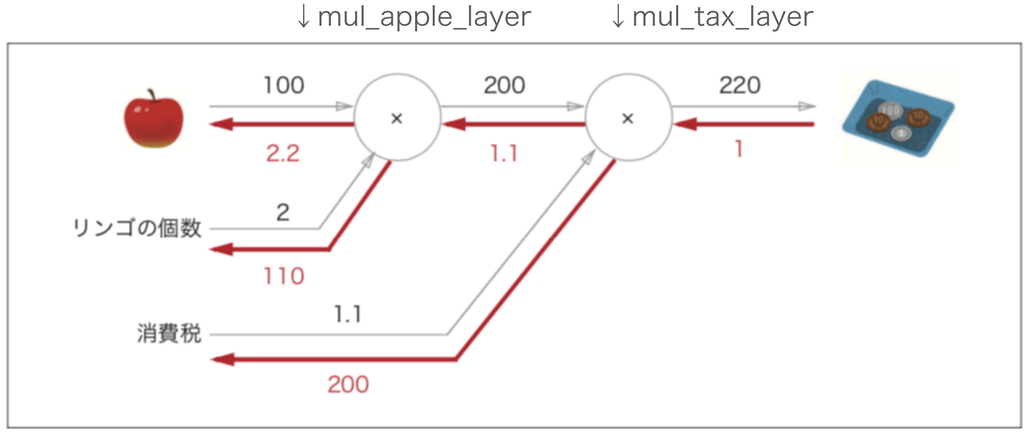


図5-16　リンゴ２個の買い物

実行結果

Atom Runner: buy_apple.py

```
price: 220          支払い金額
dApple: 2.2         りんごの値段微分
dApple_num: 110     りんごの個数微分
dTax: 200           消費税微分
```

5.4.2 加算レイヤの実装

❖加算レイヤの場合は、特に初期化は必要ない

```
23  class AddLayer:
24      def __init__(self):
25          pass
26
27      def forward(self, x, y):
28          out = x + y
29
30          return out
31
32      def backward(self, dout):
33          dx = dout * 1
34          dy = dout * 1
35
36          return dx, dy
37
```

ch05/layer_naive.py

❖backward()で上流から流れてきた微分(dout)をそのまま下流に流す

5.4.2 加算レイヤの実装

❖加算レイヤと乗算レイヤを使って「りんご２個とみかん３個の買い物」を実装する

```python
# coding: utf-8
from 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)

```

ch05/buy_apple_orange.py

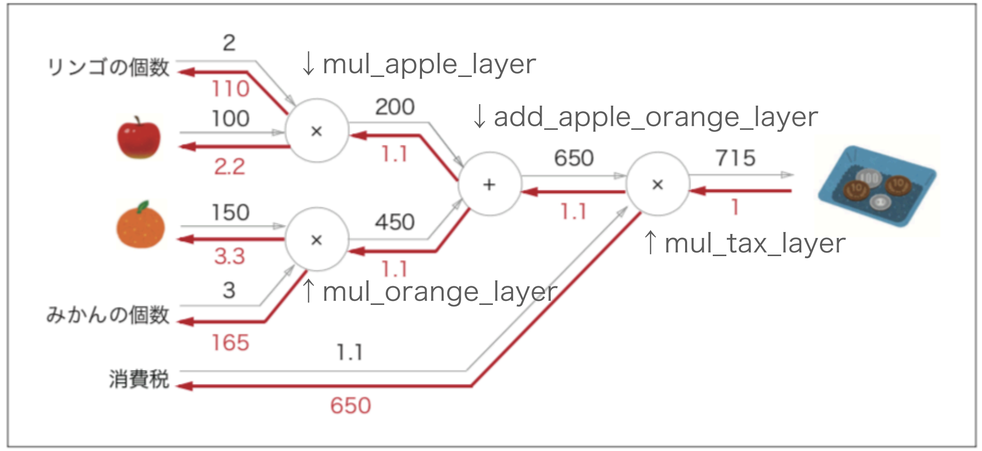


図5-17　リンゴ２個とみかん３個の買い物

実行結果

Atom Runner: buy_apple_orange.py

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

5.5 活性化関数レイヤの実装

5.5.1 RELUレイヤ

❖計算グラフの考え方をニューラルネットワークに適用する
　活性化関数であるReLUとSigmoidレイヤを実装していく
❖ReLU(Rectified Linear Unit)は次の式で表される

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

ここからxに関するyの微分は次のようになる

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

→xが0より大きければ上流の値を下流にそのまま流す
　xが0以下であれば、逆伝播はそこでストップする

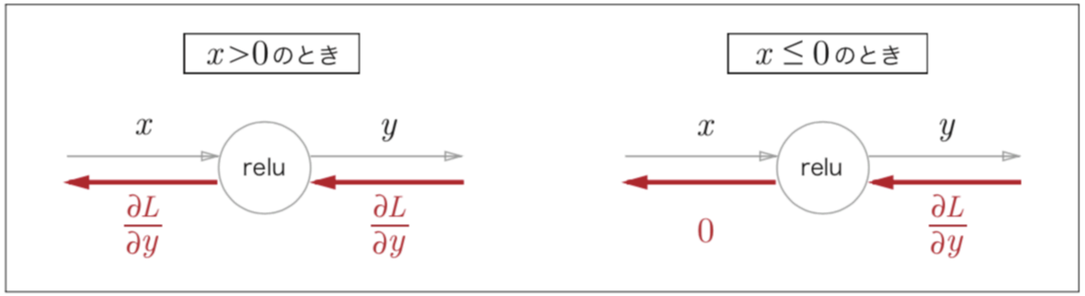


図5-18　ReLU レイヤの計算グラフ

5.5.1 RELUレイヤ

❖ReLUレイヤの実装を行う

```python
# coding: utf-8
import numpy as np
from common.functions import *
from common.util import im2col, col2im


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

❖Reluクラスは、maskという変数をインスタンス変数としてもつ

❖mask変数…True/Falseからなる Numpy配列

- xの要素で0以下の場所：True
- それ以外(0より大きい要素)：False

❖逆伝播では上流から伝播されたdoutに対してmaskの要素がTrueの場所を0に設定する

例：mask変数

```python
import numpy as np

def main():
    x = np.array([[1.0,-0.5],[-2.0,3.0]])
    print("x:",x)

    mask = (x<=0)
    print("mask:",mask)

main()
```

Atom Runner: mask.py

```
x: [[ 1.  -0.5]
 [-2.   3. ]]
mask: [[False  True]
 [ True False]]
```

5.5.2 SIGMOIDレイヤ

❖シグモイド関数の実装

シグモイド関数は次のように表される

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

計算グラフで表す

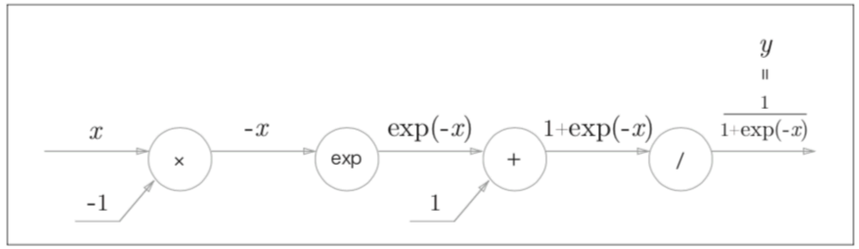


図5-19　Sigmoid レイヤの計算グラフ（順伝播のみ）

→”/”：$y = \frac{1}{x}$の計算を行う

逆伝播を考えていく

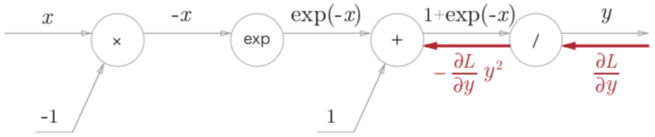
5.5.2 SIGMOIDレイヤ

❖ステップ１

$y = \frac{1}{x}$ の微分を考えていく

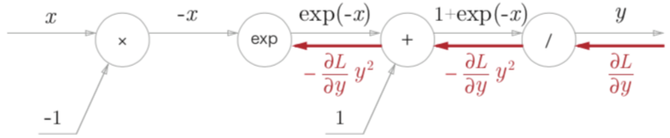
$$\frac{\partial y}{\partial x} = -\frac{1}{x^2}$$
$$= -y^2$$

つまり上流の値に対して$-y^2$を乗算して下流へ伝播する



❖ステップ２

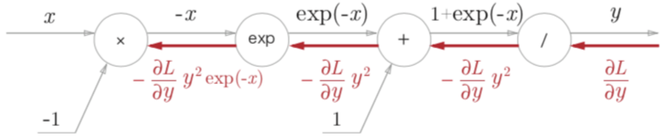
「+」のノード



❖ステップ３

「exp」ノードで、$y = \exp(x)$の微分を考えていく

$$\frac{\partial y}{\partial x} = \exp(x)$$



❖ステップ４

「×」のノード

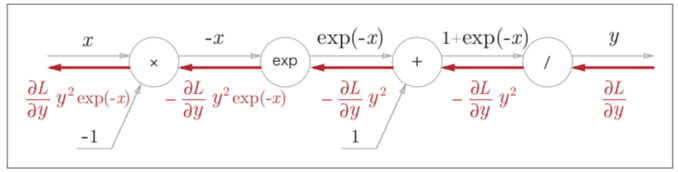


図5-20　Sigmoid レイヤの計算グラフ

→逆伝搬の出力：$\frac{\partial y}{\partial x} y^2 \exp(-x)$

（順伝播の入力xと出力yだけから計算できる）

5.5.2 SIGMOIDレイヤ

❖「sigmoid」ノード

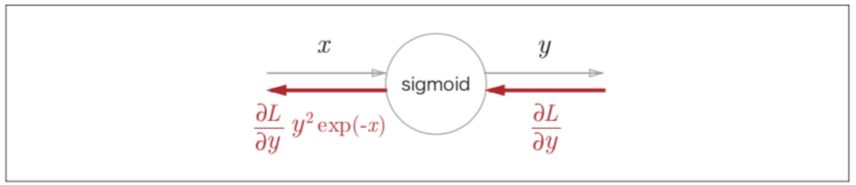


図5-21　Sigmoid レイヤの計算グラフ（簡略版）

さらに $\frac{\partial y}{\partial x} y^2 \exp(-x)$ を整理して書くことができる

$$
\begin{aligned}
\frac{\partial y}{\partial x} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1+\exp(-x))^2} \exp(-x) \\
&= \frac{\partial L}{\partial y} \frac{1}{1+\exp(-x)} \frac{\exp(-x)}{1+\exp(-x)} \\
&= \frac{\partial L}{\partial y} y(1-y)
\end{aligned}
$$

つまり逆伝播は順伝播の出力のみで計算を行うことができる

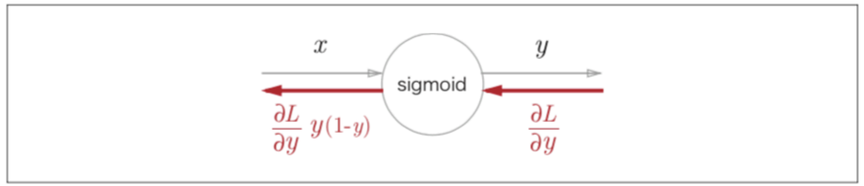


図5-22　Sigmoid レイヤの計算グラフ：順伝播の出力 y によって、逆伝播の計算を行うことができる

5.5.2 SIGMOIDレイヤ

❖SigmoidレイヤをPythonで実装する

・順伝播の出力をインスタンス変数のoutに保持する

```python
import numpy as np
from common.functions import *
from common.util import im2col, col2im
```

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

    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out

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

        return dx
```

common/layers.py

common/functions.py

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

→逆伝播時にはout変数を使って $\frac{\partial L}{\partial y} y(1-y)$ を計算する

5.6 AFFINE/SOFTMAXレイヤの実装
5.6.1 AFFINEレイヤ

❖ニューラルネットワークの順伝播では、重み付き信号の総和を計算するため、行列の積(Numpyではnp.dot())を用いた
❖この行列の積を幾何学の分野では「アフィン変換」と呼ばれる
❖アフィン変換を行う処理を「Affineレイヤ」という名前で実装していく

・行列の積とバイアスの和(np.dot(X,W) + B)を計算グラフで表す
・行列の積を計算するノードを「dot」として表す

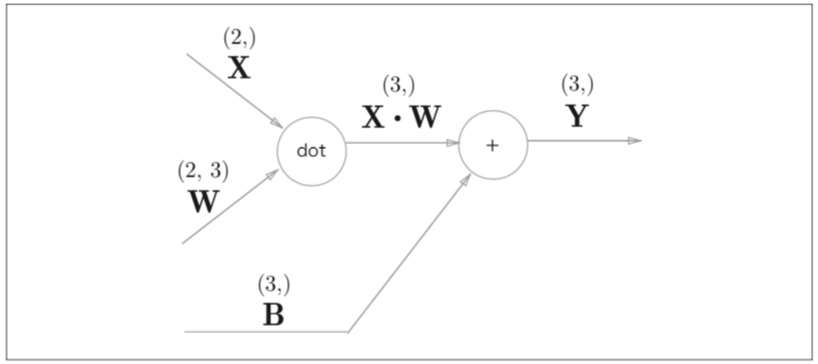


図5-24　Affine レイヤの計算グラフ：変数が行列であることに注意。各変数の上部に、その変数の形状を示す

→**X,W,B**は行列(多次元配列)

❖これまでは「スカラ値」がノード間を流れたが、「行列」がノード間を伝播する

5.6.1 AFFINEレイヤ

❖逆伝播について考えていく

行列の要素ごとに書き下す▷これまでのスカラ値と同じ手順で考えることができる

計算グラフから微分は次のようになることがわかる

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

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

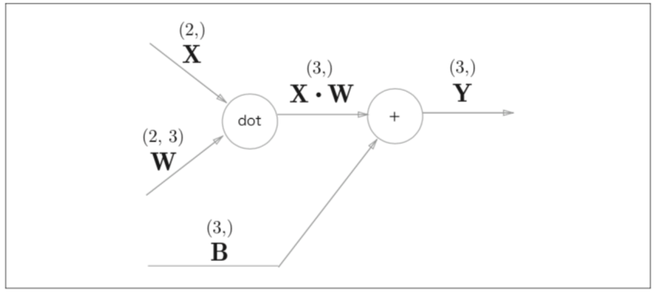


図5-24　Affine レイヤの計算グラフ：変数が行列であることに注意。各変数の上部に、その変数の形状を示す

$$\mathbf{X} = (x_0, x_1, \cdots, x_n)$$

$$\frac{\partial L}{\partial \mathbf{X}} = \left(\frac{\partial L}{\partial x_0}, \frac{\partial L}{\partial x_1}, \cdots, \frac{\partial L}{\partial x_n} \right)$$

これらは同じ形状、そのため転置が必要になる

❖Tは転置(Wの(i,j)の要素を(i,i)の要素を入れ変えることをいう)

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

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

→形状が(2,3)であったのが、転置を行うと(3,2)になる

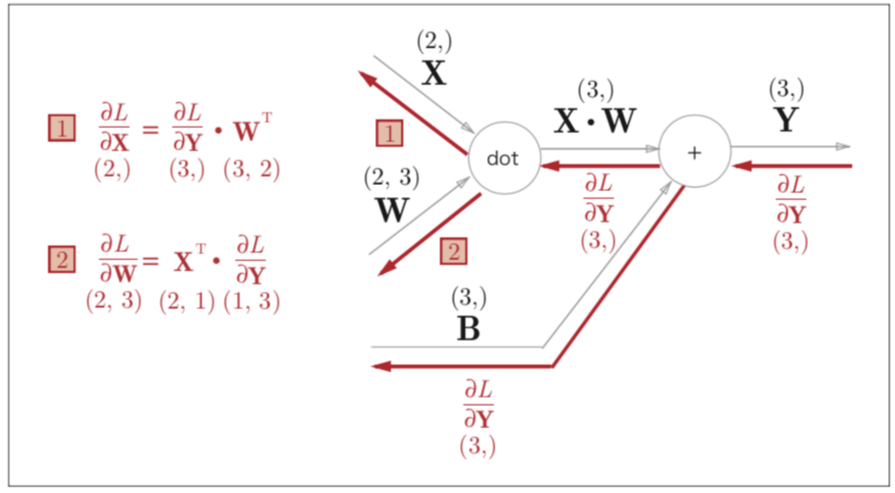


図5-25　Affine レイヤの逆伝播：変数が多次元配列であることに注意。逆伝播の際の各変数の下部に、その変数の形状を示す

5.6.2 バッチ版AFFINEレイヤ

❖これまで：Affineレイヤの入力**X**は一つのデータを対象とした

❖N個のデータをまとめて順伝播する場合、バッチ版のAffineレイヤを考える

・計算グラフは次のようになる

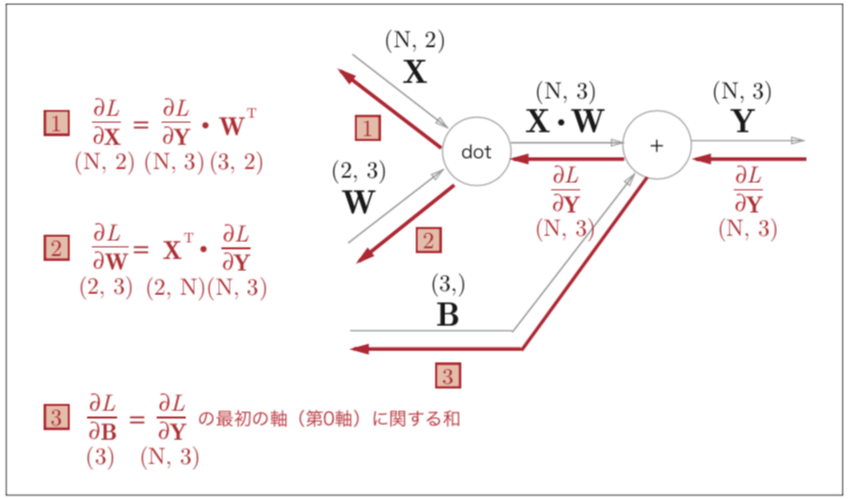


図5-27　バッチ版 Affine レイヤの計算グラフ

→入力である**X**の形状が(N,2)になっただけ

　バイアスの加算に対してはそれぞれのデータに加算されることに注意

5.6.2 バッチ版AFFINEレイヤ

❖Affineの実装は次のようになる

❖入力データがテンソル(4次元のデータ)の場合を考慮している

```python
class Affine:
    def __init__(self, W, b):
        self.W =W
        self.b = b

        self.x = None
        self.original_x_shape = None
        # 重み・バイアスパラメータの微分
        self.dW = None
        self.db = None

    def forward(self, x):
        # テンソル対応
        self.original_x_shape = x.shape
        x = x.reshape(x.shape[0], -1)
        self.x = x

        out = np.dot(self.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)

        dx = dx.reshape(*self.original_x_shape)  # 入力データの形状に戻す（テンソル対応）
        return dx
```

common/layers.py

5.6.3 SOFTMAX-WITH-LOSSレイヤ

❖出力層であるソフトマックス関数について考えていく
❖ソフトマックス関数：入力された値を正規化して出力する(出力の和が1になるように変形する)

例：手書き数字認識の場合のSoftmaxレイヤの出力(10クラス分類)

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

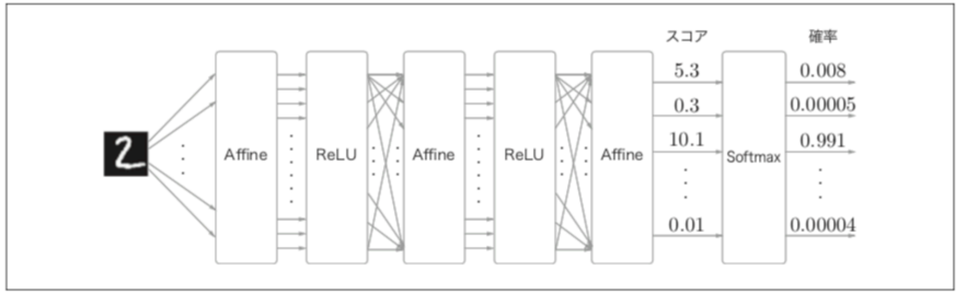


図5-28　入力画像が、Affine レイヤと ReLU レイヤによって変換され、Softmax レイヤによって 10 個の入力が正規化される。この例では、「0」であるスコアは 5.3 であり、これが Softmax レイヤによって 0.008（0.8%）に変換される。また、「2」であるスコアは 10.1 であり、これは 0.991（99.1%）に変換される

❖ニューラルネットワークで行う処理 - **推論**と**学習**の２つに分かれる
　推論では、通常Softmaxレイヤは使用しない
　正規化しない出力結果のことを「スコア」と呼ぶことがある→推論ではスコアの最大値にのみ興味があるため

5.6.3 SOFTMAX-WITH-LOSSレイヤ

❖Softmaxレイヤを実装していく
損失関数である交差エントロピーも含めて「Softmax-with-Lossレイヤ」という名前のレイヤで実装をする
計算グラフは次のよう(3クラス分類)

t：教師ラベル

❖逆伝播の結果は、出力と教師ラベルの差分になる
→このようになるように、交差エントロピーは設計された

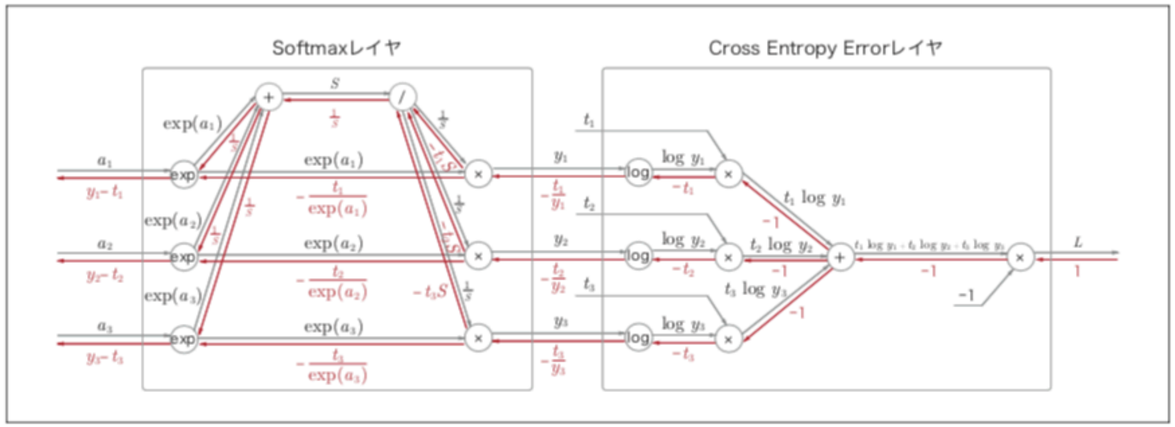


図5-29　Softmax-with-Loss レイヤの計算グラフ

導出過程は付録A

簡略化

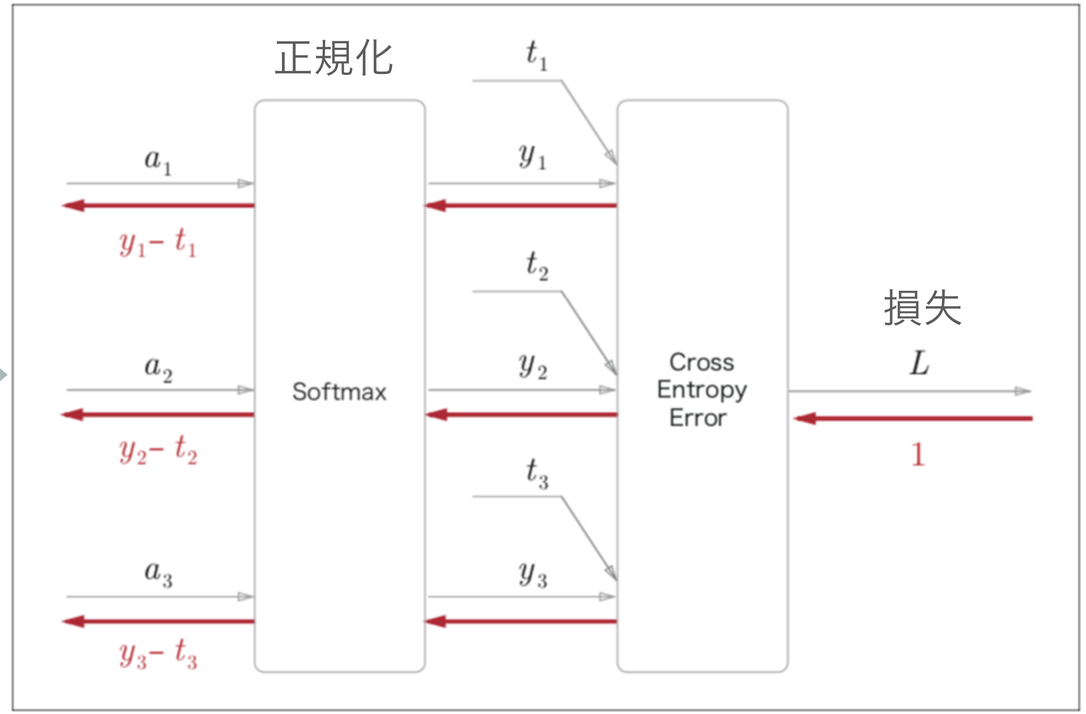


図5-30　「簡易版」Softmax-with-Loss レイヤの計算グラフ

5.6.3 SOFTMAX-WITH-LOSSレイヤ

具体例：教師ラベルが(0,1,0)であるデータに対して、Softmaxレイヤの出力が(0.3,0.2,0.5)であった場合
→正解ラベルに対する確率は0.2(20%)であるため、ニューラルネットワークは正しい認識ができていない。

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

    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]
        if self.t.size == self.y.size: # 教師データがone-hot-vectorの場合
            dx = (self.y - self.t) / batch_size
        else:
            dx = self.y.copy()
            dx[np.arange(batch_size), self.t] -= 1
            dx = dx / batch_size

        return dx
```

common/layers.py

・Softmaxレイヤからの逆伝播は、(y-t)より、(0.3,-0.8,0.5)という大きな誤差を伝播することになる

→その大きな誤差から大きな内容を学習することになる

別の具体例：教師ラベルが(0,1,0)であるデータに対して、Softmaxレイヤの出力が(0.01,0.99,0)の場合

・Softmaxからの逆伝播は(0.01,-0.01,0)という小さな誤差になる

→前レイヤが学習する内容も小さくなる

❖この実装ではソフトマックスと交差エントロピー誤差の実装

❖バッチサイズで割ることでデータ１個あたりの誤差が前レイヤへ伝播する

5.7 誤差逆伝播法の実装

5.7.1 ニューラルネットワークの学習の全体図

❖ニューラルネットワークの学習の全体図

■**前提**

ニューラルネットワークは、重みとバイアスがあり、この重みとバイアスを訓練データに適応するように学習する

学習は次の４つの手順で行う

◉**ステップ１(ミニバッチ)**

訓練データの中からランダムに１部のデータを選び出す

◉**ステップ２(勾配の算出)**

各重みパラメータに関する損失関数の勾配を求める

誤差逆伝播法が登場するのはここ！

◉**ステップ３(パラメータの更新)**

重みパラメータを勾配方向に微小量だけ更新する

◉**ステップ４(繰り返す)**

ステップ１、ステップ２、ステップ３を繰り返す

☹前章では、勾配を求めるために数値微分を利用した。多くの時間が計算にかかる。

☻誤差逆伝播法では高速に効率良く勾配を求めることができる

5.7.2誤差逆伝播法に対応したニューラルネットワークの実装

❖２層のニューラルネットワークをTwoLayerNetとして実装していく
　このクラスのインスタンス変数とメソッドを整理すると次の表のようになる
❖前章の「4.5 学習アルゴリズムの実装」と共通する部分が多くある。変更箇所はレイヤを使用していること。

表5-1　TwoLayerNet クラスのインスタンス変数

インスタンス変数	説明
params	ニューラルネットワークのパラメータを保持するディクショナリ変数。 params['W1'] は１層目の重み、params['b1'] は１層目のバイアス。 params['W2'] は２層目の重み、params['b2'] は２層目のバイアス。
layers	ニューラルネットワークのレイヤを保持する**順番付きディクショナリ**変数。 layers['Affine1']、layers['Relu1']、layers['Affine2'] といったように**順番付きディクショナリ**で各レイヤを保持する。
lastLayer	ニューラルネットワークの最後のレイヤ。 この例では、SoftmaxWithLoss レイヤ。

表5-2　TwoLayerNet クラスのメソッド

メソッド	説明
__init__(self, input_size, hidden_size, output_size, weight_init_std)	初期化を行う。 引数は頭から順に、入力層のニューロンの数、隠れ層のニューロンの数、出力層のニューロンの数、重み初期化時のガウス分布のスケール。
predict(self, x)	認識（推論）を行う。 引数の x は画像データ。
loss(self, x, t)	損失関数の値を求める。 引数の x は画像データ、t は正解ラベル。
accuracy(self, x, t)	認識精度を求める。
numerical_gradient(self, x, t)	重みパラメータに対する勾配を数値微分によって求める（前章と同じ）。
gradient(self, x, t)	重みパラメータに対する勾配を誤差逆伝播法によって求める。

5.7.2誤差逆伝播法に対応したニューラルネットワークの実装

❖TwoLayerNetの実装

```python
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from common.layers import *
from common.gradient import numerical_gradient
from collections import OrderedDict


class TwoLayerNet:
    ＊初期化を行う
    def __init__(self, input_size, hidden_size, output_size, weight_init_std = 0.01):
        # 重みの初期化
        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)

        # レイヤの生成
        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()
    ＊認識（推論）を行う
    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x
    ＊損失関数の値を求める
    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)
    ＊認識精度を求める
    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
    ＊重みパラメータに対する勾配を数値微分によって求める
    # x:入力データ, t:教師データ
    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
    ＊重みパラメータに対する勾配を誤差逆伝播法によって求める
    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)
```

common/two_layer_net.py

→レイヤを
OrderedDictで保持
(順番付き)

5.7.3 誤差逆伝播法の勾配確認

❖これまでの勾配を求める方法

⑴数値微分によって求める方法　⑵解析的に数式を解いて求める方法

⑵で誤差逆伝播法を用いることで大量のパラメータが存在しても効率よく計算を行うことができた

⑴は、⑵の実装の正しさを確認する場面で使用する(⑴の方が実装が簡単である→ミスが起きにくい)

❖勾配確認…数値微分で勾配を求めた結果と、誤差逆伝播法で求めた結果が一致すること（ほぼ近いこと）を確認する作業

・勾配確認の実装

```python
# coding: utf-8
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# データの読み込み
(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)

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

ch05/gradient_check.py

実行結果

Atom Runner: gradient_check.py

```
W1:4.6131484514530405e-10
b1:2.555988176198426e-09
W2:6.359008743551535e-09
b2:1.4090341207279035e-07
```

→毎回違う！？

→数値微分と誤差逆伝播法でそれぞれ求めた勾配の差はかなり小さい=誤差逆伝播法で求めた勾配も正しい結果である

5.7.4 誤差逆伝播法を使った学習

❖誤差逆伝播法を使ったニューラルネットワークの学習の実装

```python
# coding: utf-8
import sys, os
sys.path.append(os.pardir)

import numpy as np
from dataset.mnist import load_mnist
from two_layer_net import TwoLayerNet

# データの読み込み
(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]

    # 勾配
    #grad = network.numerical_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)
```

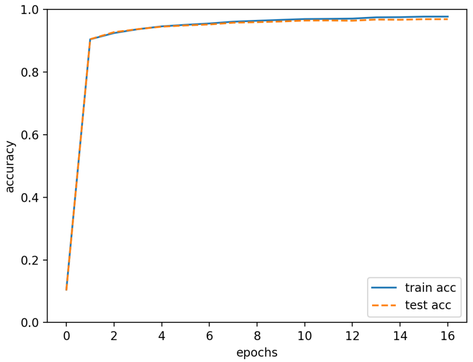
```python
    # 更新
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]

    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(train_acc)
        test_acc_list.append(test_acc)
        print(train_acc, test_acc)
```

実行結果

Atom Runner: train_neuralnet.py

```
0.10478333333333334 0.1025
0.9046 0.905
0.9246666666666666 0.9275
0.9369166666666666 0.9371
0.94605 0.945
0.95095 0.9489
0.9552333333333334 0.9521
0.9609333333333333 0.9577
0.9639166666666666 0.9594
0.9667333333333333 0.9615
0.9693833333333334 0.9646
0.9699666666666666 0.9649
0.9708333333333333 0.9641
0.9748666666666667 0.9677
0.9754 0.9671
0.97715 0.9688
0.9772333333333333 0.9688
```

認識精度



→過学習は起きていない！

ch05/train_neuralnet.py

5.8 まとめ

❖視覚的に計算の過程を表す計算グラフという方法を学んだ

❖ニューラルネットワークで使用する誤差逆伝播法を説明した

❖行う処理をレイヤという単位で実装した

（ReLU、Softmax-with-Loss、Affine、Softmaxレイヤ）

❖forward(順方向)、backward(逆方向)にデータを伝播することで、重みパラメータの勾配を効率的に求めることができる

❖数値微分と誤差逆伝播法の結果を比較することで、誤差逆伝播法の実装に誤りがないことを確認できる(勾配確認)

付録A

付録A SOFTMAX-WITH-LOSSレイヤの計算グラフ

❖ソフトマックス関数と交差エントロピー誤差の計算グラフを示し、逆伝播を求める
❖ソフトマックス関数はSoftmaxレイヤ、交差エントロピー誤差はCross Entropy Errorレイヤと呼ぶ
❖この２つの組み合わせたレイヤをSoftmax-with-Lossレイヤと呼ぶ

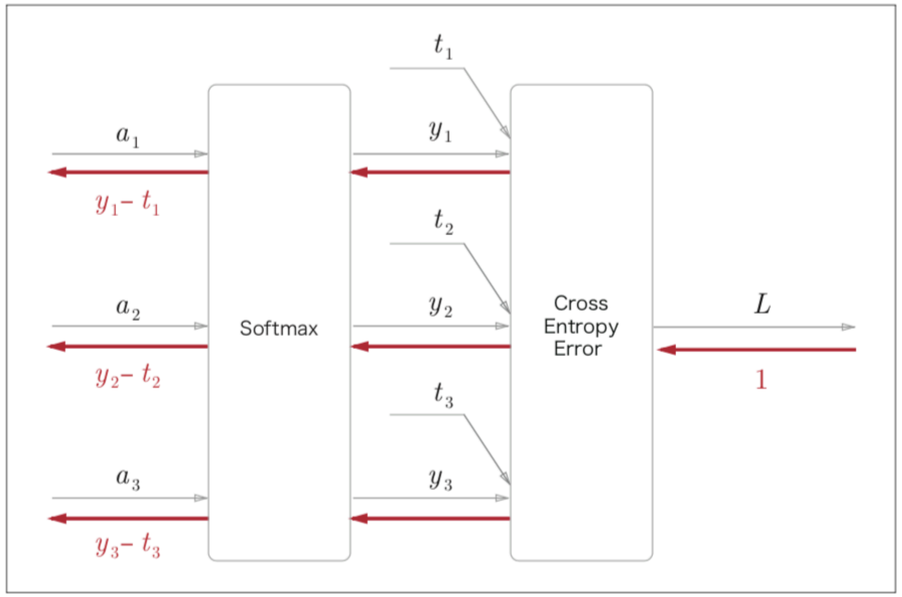


図 A-1　Softmax-with-Loss レイヤの計算グラフ

・3クラス分類を行うニューラルネットワークを想定

前レイヤからの入力：(a_1, a_2, a_3)

Softmaxレイヤの出力：(y_1, y_2, y_3)

教師ラベル：(t_1, t_2, t_3)

Cross Entropy Errorレイヤの出力：損失L

❖Softmax-with-Lossレイヤの逆伝播の結果：$(y_1 - t_1, y_2 - t_2, y_3 - t_3)$

A.1 順伝播

❖ソフトマックス関数

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

・計算グラフでのSはソフトマックス関数の分母に当たる

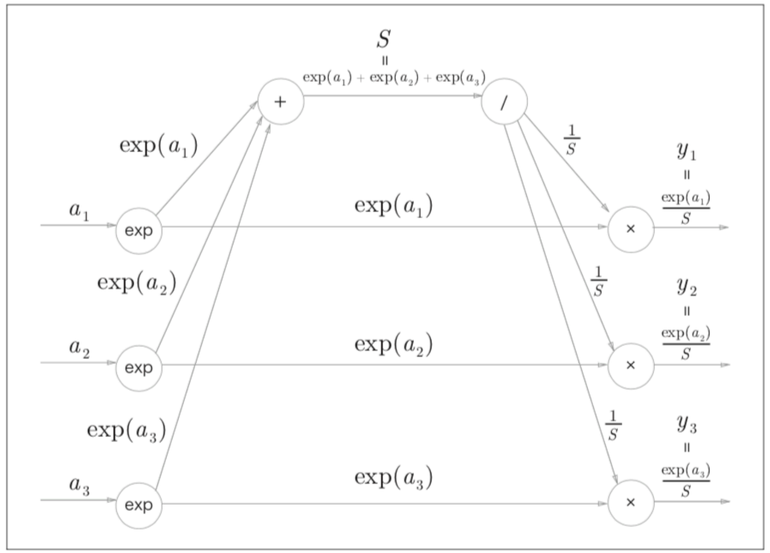


図 A-2　Softmax レイヤの計算グラフ（順伝播のみ）

❖交差エントロピー誤差

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

・純粋に式を計算グラフに表した

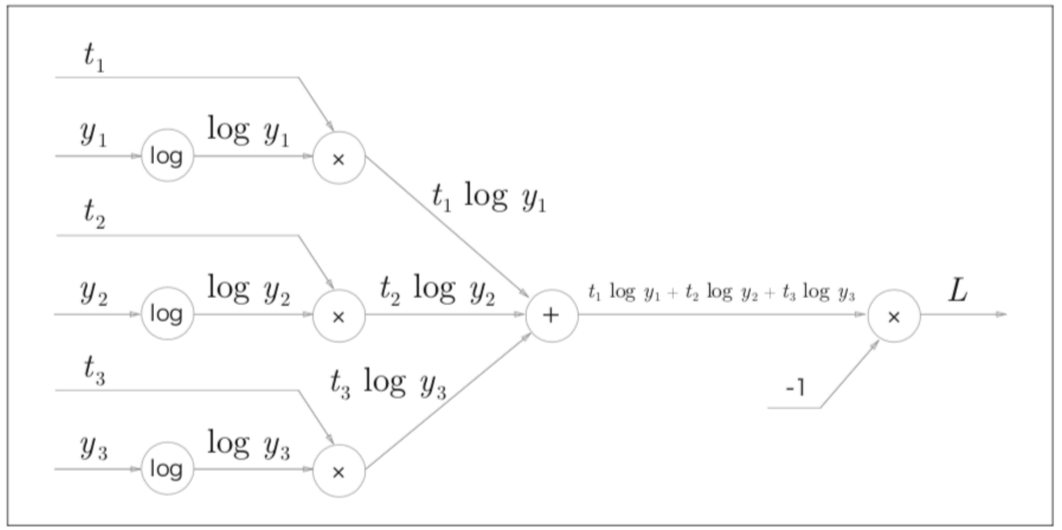


図 A-3　Cross Entropy Error レイヤの計算グラフ（順伝播のみ）

A.2 逆伝播

❖Cross Entropy Errorレイヤの逆伝播

・逆伝播を求める際に、気をつける点

✦逆伝播の最初の値(計算グラフの一番右の値)は1である

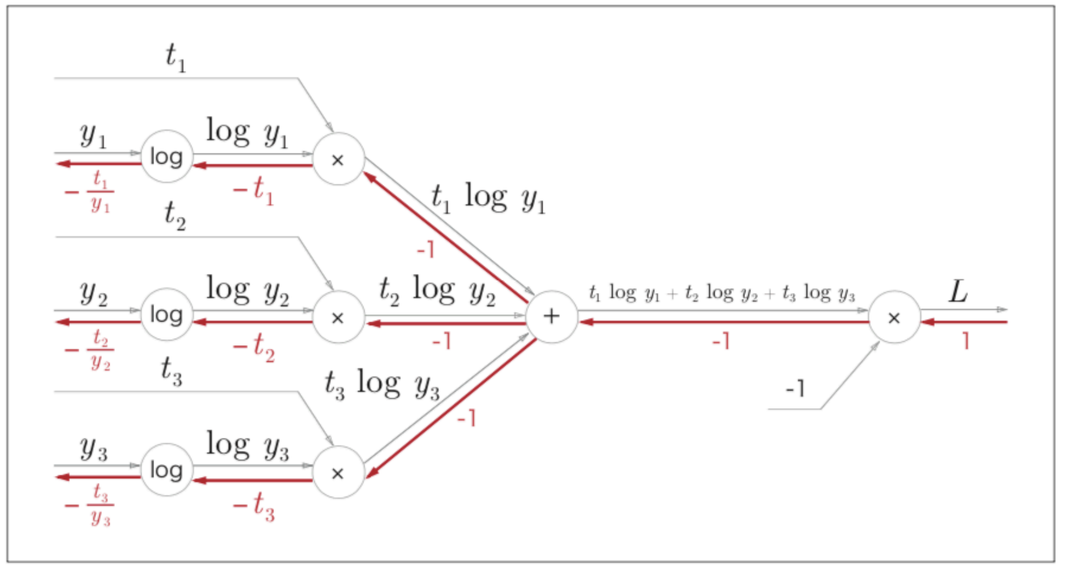
✦「×」ノードの逆伝播では順伝播の時の入力値のひっくり返した値を上流からの微分に対して乗算して下流に流す

✦「+」ノードでは上流から伝わる値をそのまま流す

✦「log」ノードの逆伝播は次の式に従う

$$
\begin{aligned}
y &= \log x \\
\frac{\partial y}{\partial x} &= \frac{1}{x}
\end{aligned}
$$

❖結果(Softmaxレイヤへの逆伝播の入力)：$(-\frac{t_1}{y_1}, -\frac{t_2}{y_2}, -\frac{t_3}{y_3})$

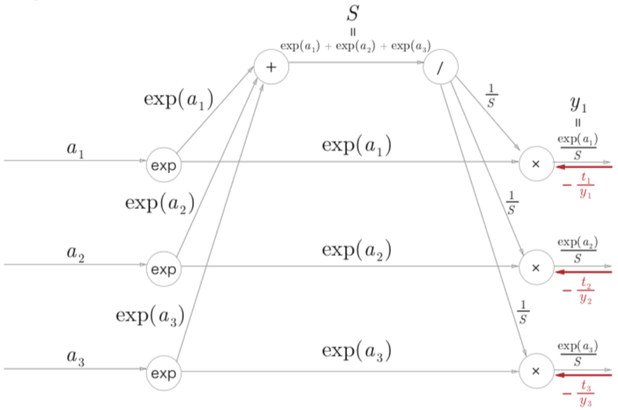


図A-4　交差エントロピー誤差の逆伝播

A.2 逆伝播

❖Siftmaxレイヤの逆伝播

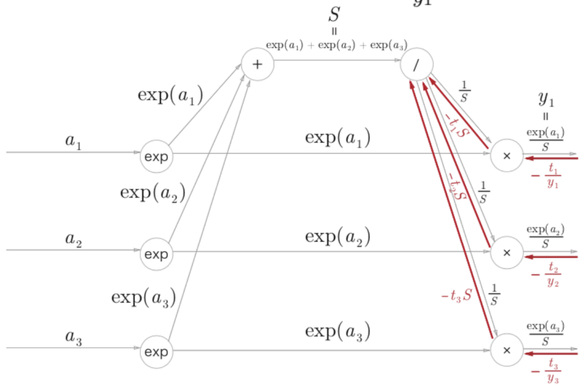
◉ステップ１



前レイヤから逆伝播の値が流れてくる

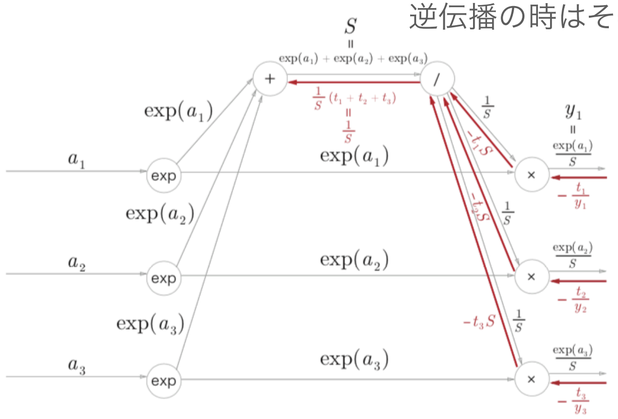
◉ステップ２

$$-\frac{t_1}{y_1}\exp(a_1) = -t_1\frac{S}{\exp(a_1)}\exp(a_1) = -t_1 S$$



◉ステップ３

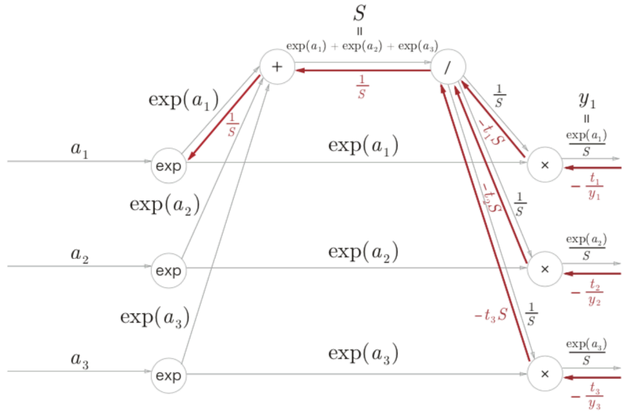
順伝播の時に複数に枝分かれして流れた場合、
逆伝播の時はその逆伝播の値が加算される



結果：$\frac{1}{S}(t_1 + t_2 + t_3)$

ちなみに、教師ラベルは「one-hotベクトル」

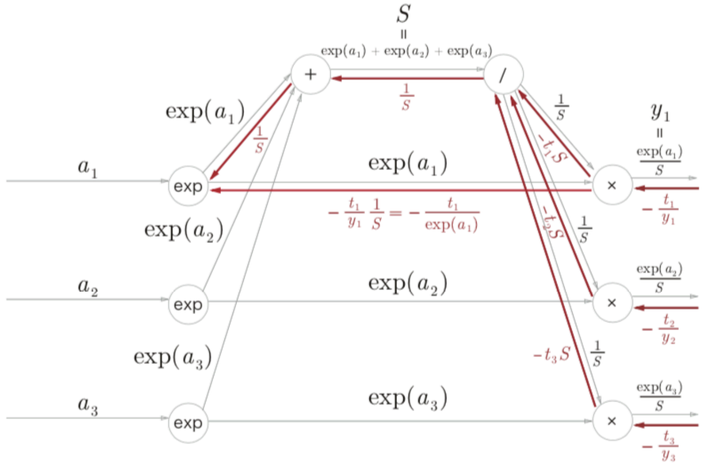
◉ステップ４



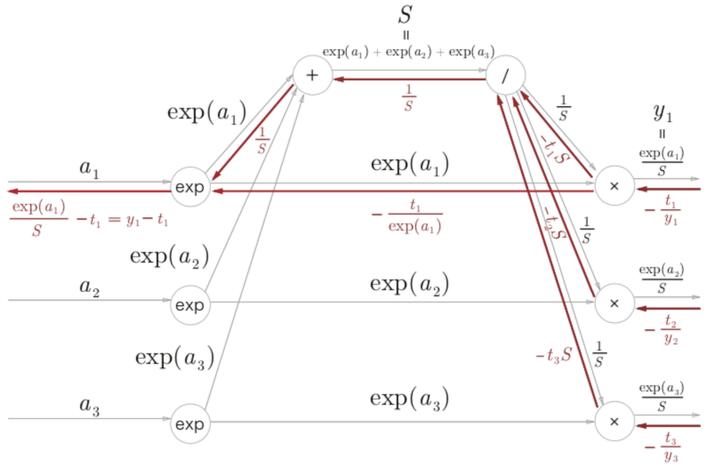
A.2 逆伝播

❖Siftmaxレイヤの逆伝播

◉ステップ5



◉ステップ6



・「exp」ノードに関して

$$
\begin{aligned}
y &= \exp(x) \\
\frac{\partial y}{\partial x} &= \exp(x)
\end{aligned}
$$

❖結果： $y_1 - t_1$

A.3 まとめ

❖Softmax-with-Lossレイヤの計算グラフ

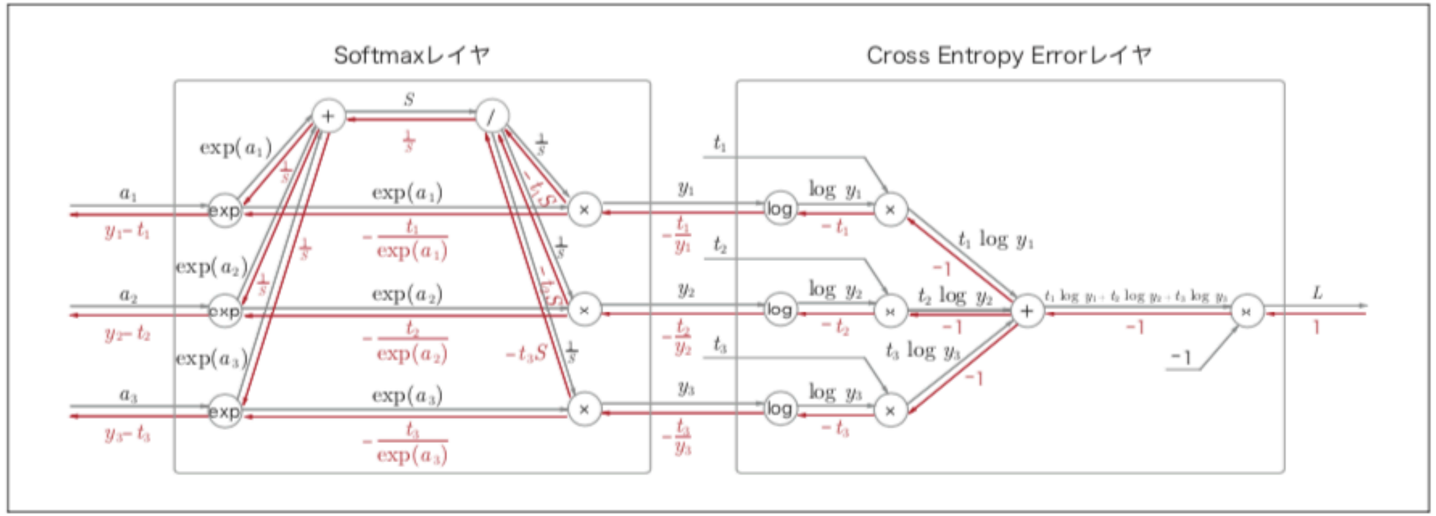


図 A-5　Softmax-with-Loss レイヤの計算グラフ

BACK UP