ゼロから作るディープラーニング
５章

担当：遠藤研

4章の復習

- 損失関数
  - ニューラルネットワークの性能の悪さを示す指標
  - 二乗和誤差
    - $E = \frac{1}{2}\sum_k (y_k - t_k)^2$
  - 交差エントロピー誤差
    - $E = -\sum_k t_k \log y_k$

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

4章の復習

- 勾配法（勾配降下法）
  - 勾配が0になる箇所を探索
  - $x = x - \eta \frac{\partial f}{\partial x}$

5.1 計算グラフ

計算グラフとは・・・

計算の過程をグラフで表したもの、複数のノードとエッジで表現

問 1：太郎くんはスーパーで 1 個 100 円のリンゴを 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。

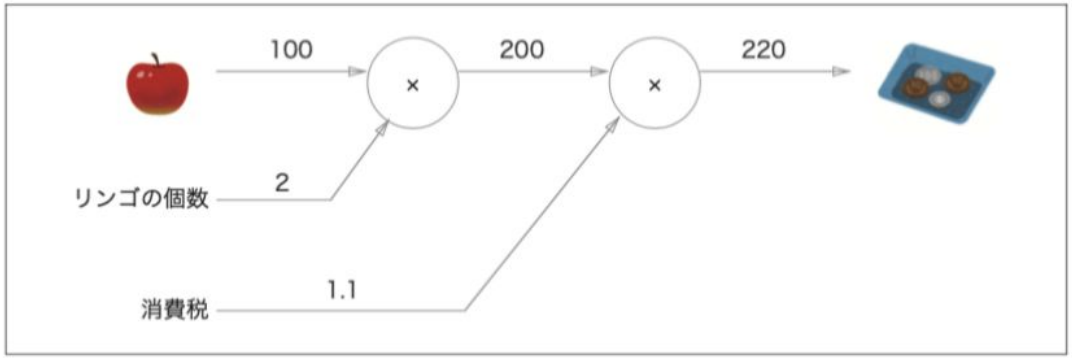


図5-2　計算グラフによる問 1 の答え：「リンゴの個数」と「消費税」を変数として、○の外に表記する

5.1 計算グラフ

問 1：太郎くんはスーパーで 1 個 100 円のリンゴを 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。

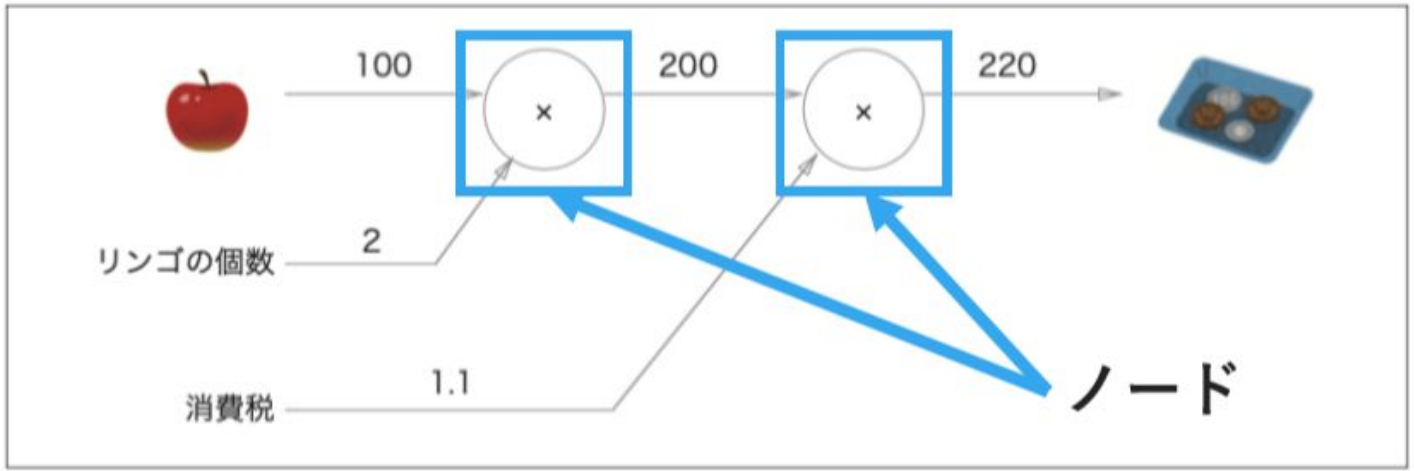


図5-2　計算グラフによる問 1 の答え：「リンゴの個数」と「消費税」を変数として、○の外に表記する

5.1 計算グラフ

問 1：太郎くんはスーパーで 1 個 100 円のリンゴを 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。

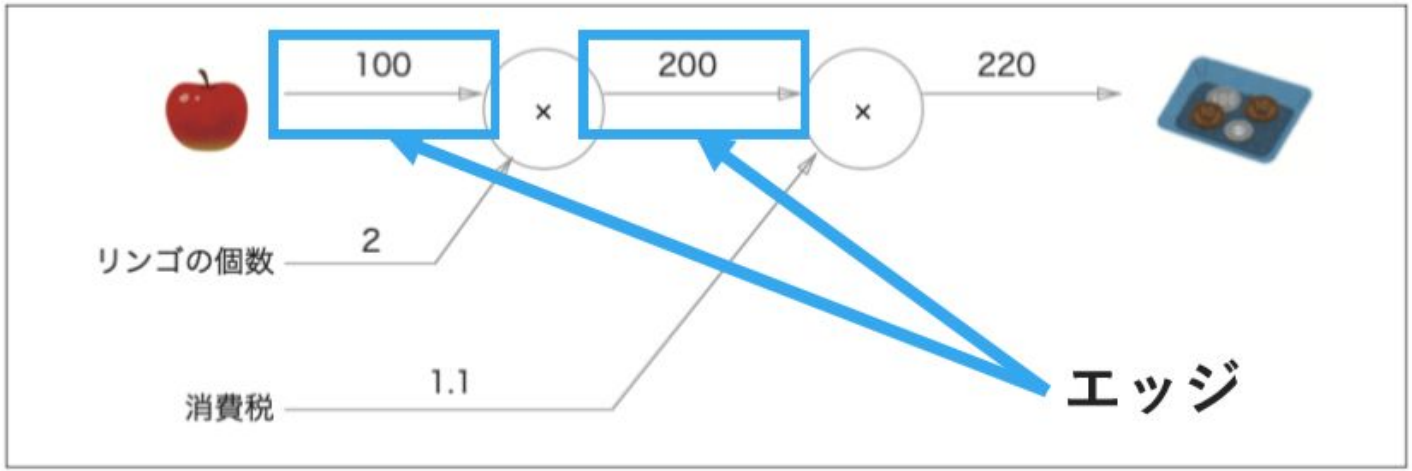


図5-2　計算グラフによる問 1 の答え：「リンゴの個数」と「消費税」を変数として、○の外に表記する

5.1 計算グラフ

問 1：太郎くんはスーパーで 1 個 100 円のリンゴを 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。

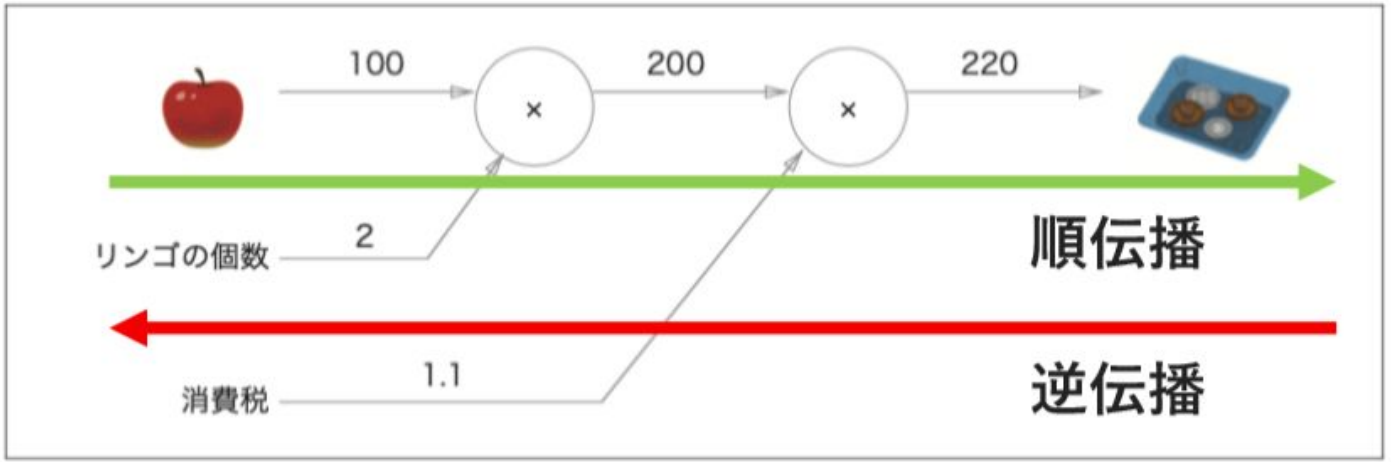


図5-2　計算グラフによる問 1 の答え：「リンゴの個数」と「消費税」を変数として、○の外に表記する

5.1 計算グラフ

問 2：太郎くんはスーパーでリンゴを 2 個、みかんを 3 個買いました。リンゴは 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。

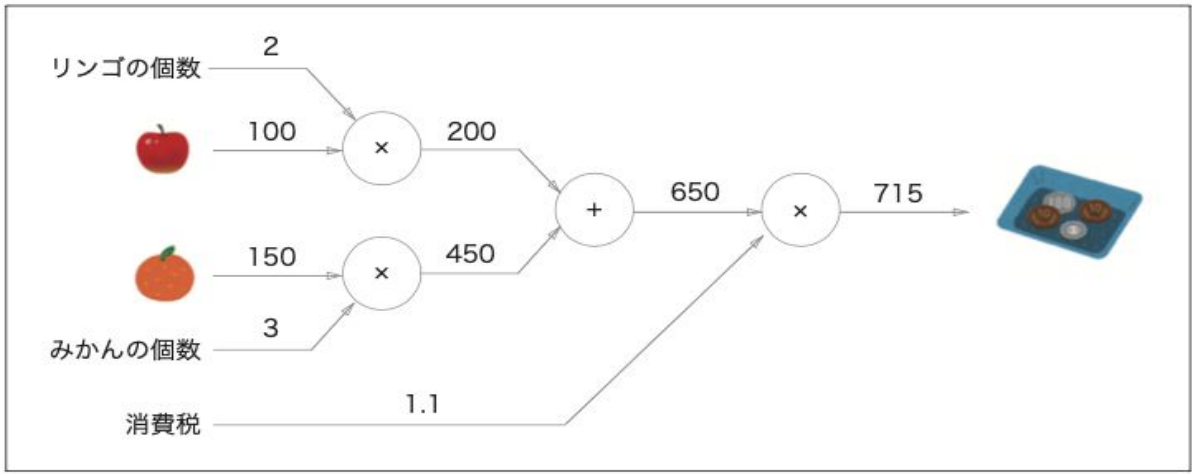


図 5-3　計算グラフによる問 2 の答え

計算グラフの利点

- **「局所的な計算」で単純な計算の組み合わせにできる**
- 逆伝播を使って「微分」を効率良く計算できる

局所的な計算

- 計算グラフを用いると「局所的な計算」を伝播することで最終的な結果を得ることができる

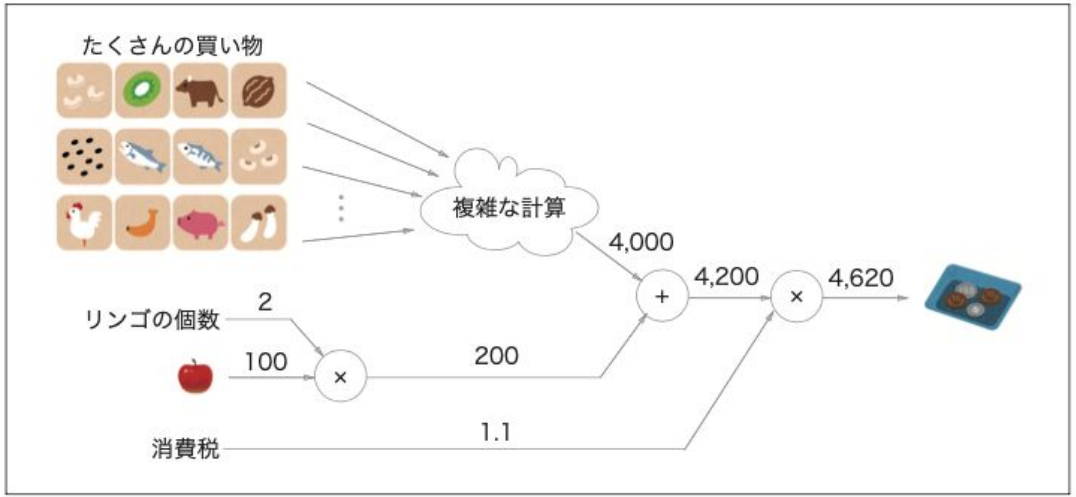


図5-4　リンゴ２個とそれ以外のたくさんの買い物の例

局所的な計算

- 計算グラフを用いると**「局所的な計算」**を伝播することで最終的な結果を得ることができる

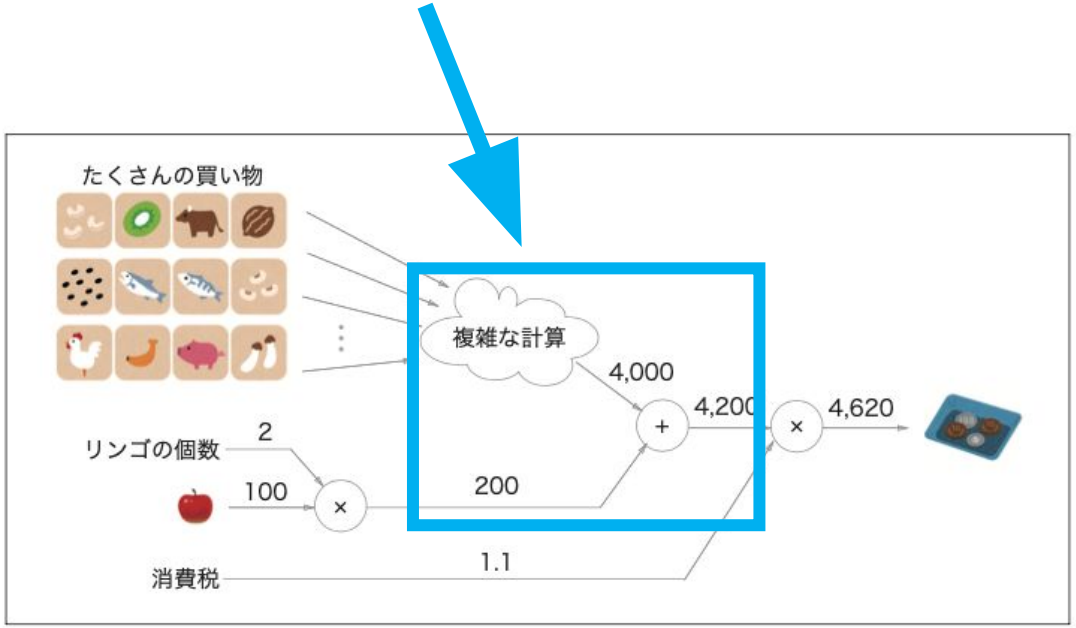


図5-4　リンゴ２個とそれ以外のたくさんの買い物の例

計算グラフの利点

- 「局所的な計算」で単純な計算の組み合わせにできる
- **逆伝播を使って「微分」を効率良く計算できる**

簡単な逆伝播

問 1：太郎くんはスーパーで 1 個 100 円のリンゴを 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。

上の問題でリンゴの値段が値上がりした場合の、最終的な支払金額への影響量を考える

- リンゴの値段をx、支払い金額をLとした場合、$\frac{\partial L}{\partial x}$を求める

簡単な逆伝播

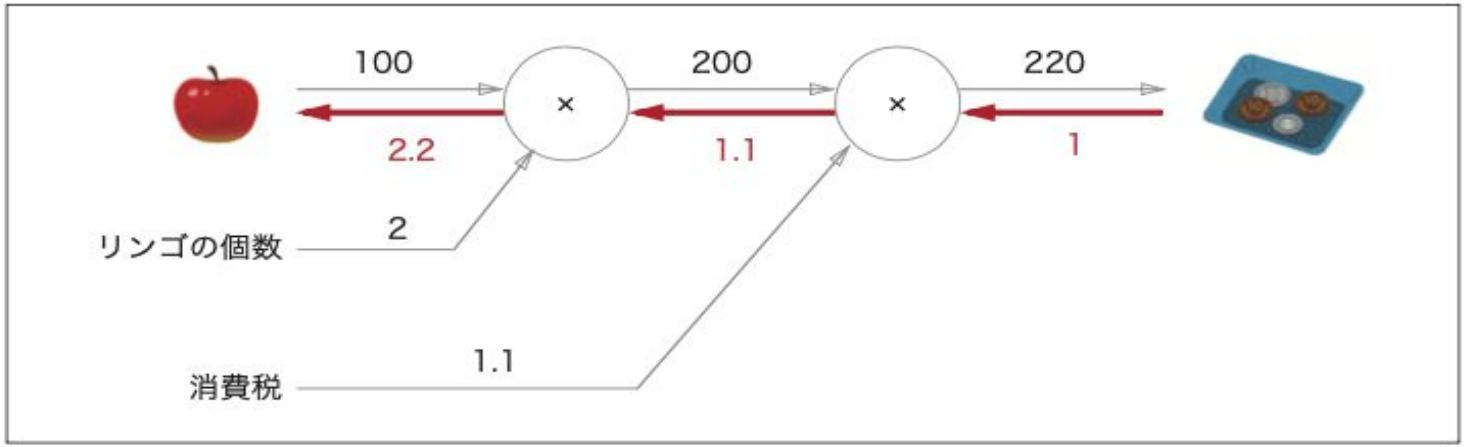


図 5-5　逆伝播による微分値の伝達

リンゴが1円値上がりした場合、最終的な支払い金額が2.2円増える

正確には、リンゴの値段がある微小な値だけ増えたら、
最終的な金額はその微小な値の 2.2 倍だけ増加することを意味する

簡単な逆伝播

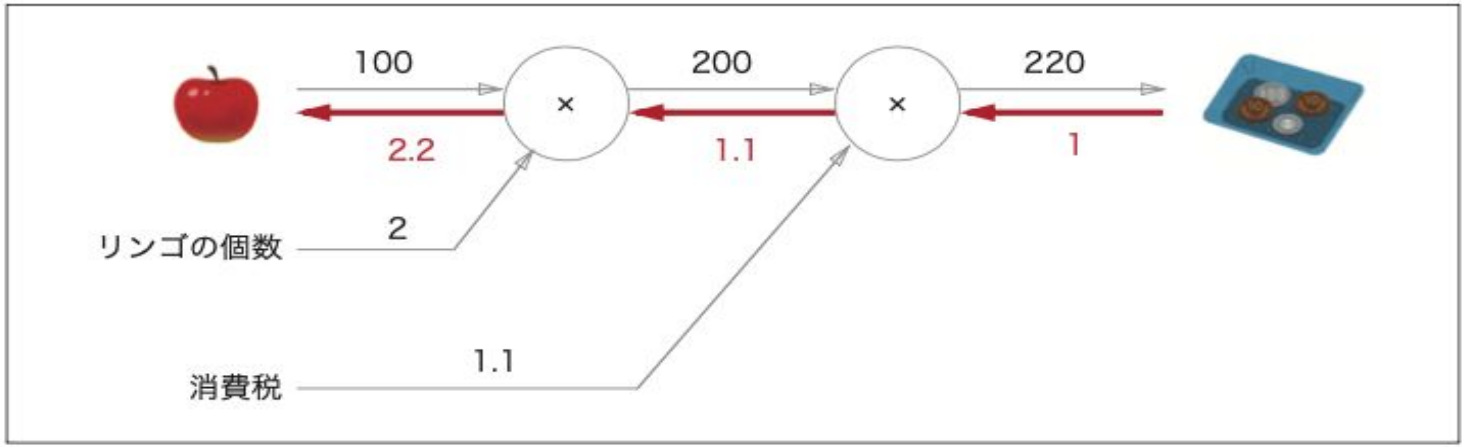


図 5-5　逆伝播による微分値の伝達

途中までの計算は共通
→計算結果を共有できるので効率が良い

5.2 連鎖率

下は、**y = f(x)** という計算の逆伝搬を表したもの

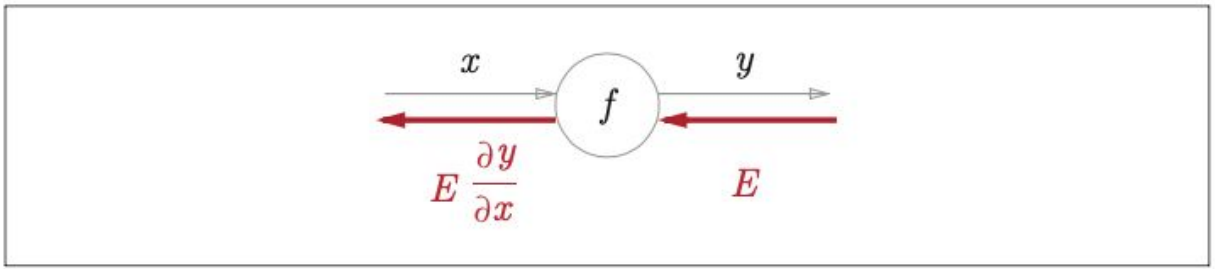


図5-6　計算グラフの逆伝播：順方向とは逆向きに、局所的な微分を乗算する

ノードの局所的な微分(∂y/∂x)を計算し、Eに乗算するだけ

→　これを繰り返して目的とする微分の値を効率よく求める

5.2 連鎖率

ここで、微分したい関数を合成関数　　$z = (x + y)^2$する

この式は2つの関数に分ける事ができる

$$z = t^2$$

$$t = x + y$$

この関数をXで微分したい

$$\frac{\partial z}{\partial x} = \cdots ?$$

5.2 連鎖率

連鎖率の定義

　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

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

5.2 連鎖率

$$z = t^2$$

$$t = x + y$$

を連鎖率を使って微分すると下のようになる。

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

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

5.2 連鎖率

さっきの微分の計算を計算グラフで表すと下のようになる

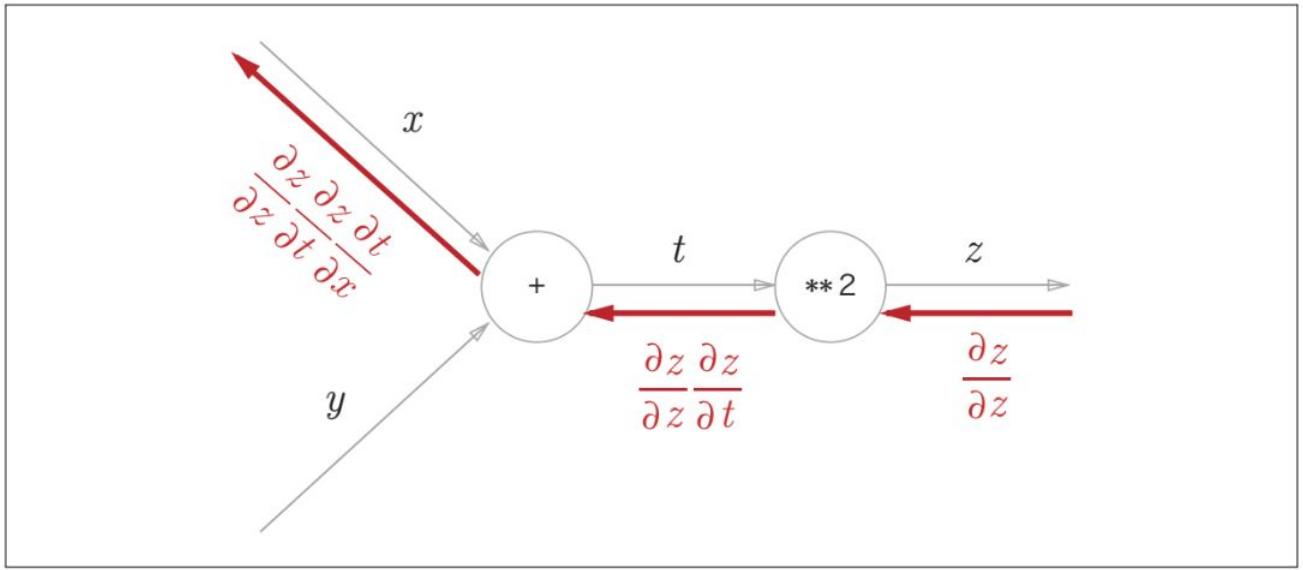


図5-7　式 (5.4) の計算グラフ：順方向とは逆向きの方向に、局所的な微分を乗算して渡していく

5.2 連鎖率

さっきの微分の計算を計算グラフで表すと下のようになる

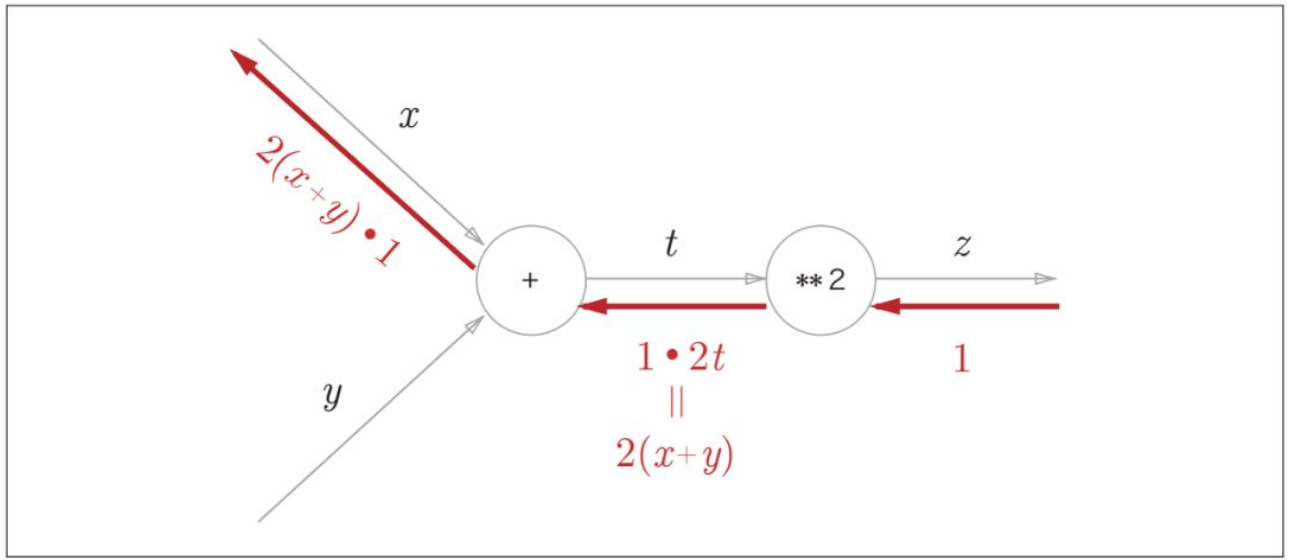


図5-8　計算グラフの逆伝播の結果より、$\frac{\partial z}{\partial x}$ は $2(x+y)$ となる

5.3 逆伝播

逆伝播の仕組みを実際の演算を例に説明していく。

5.3.1 加算ノードの逆伝播

z = x + y という数式について考えると、それぞれの偏微分は次のようになる。

$$\frac{\partial z}{\partial x} = 1$$
$$\frac{\partial z}{\partial y} = 1 \qquad (5.5)$$

逆伝播では、上流から伝わる値の微分に、1を乗算して下流に流す。

計算グラフは次のようになる。

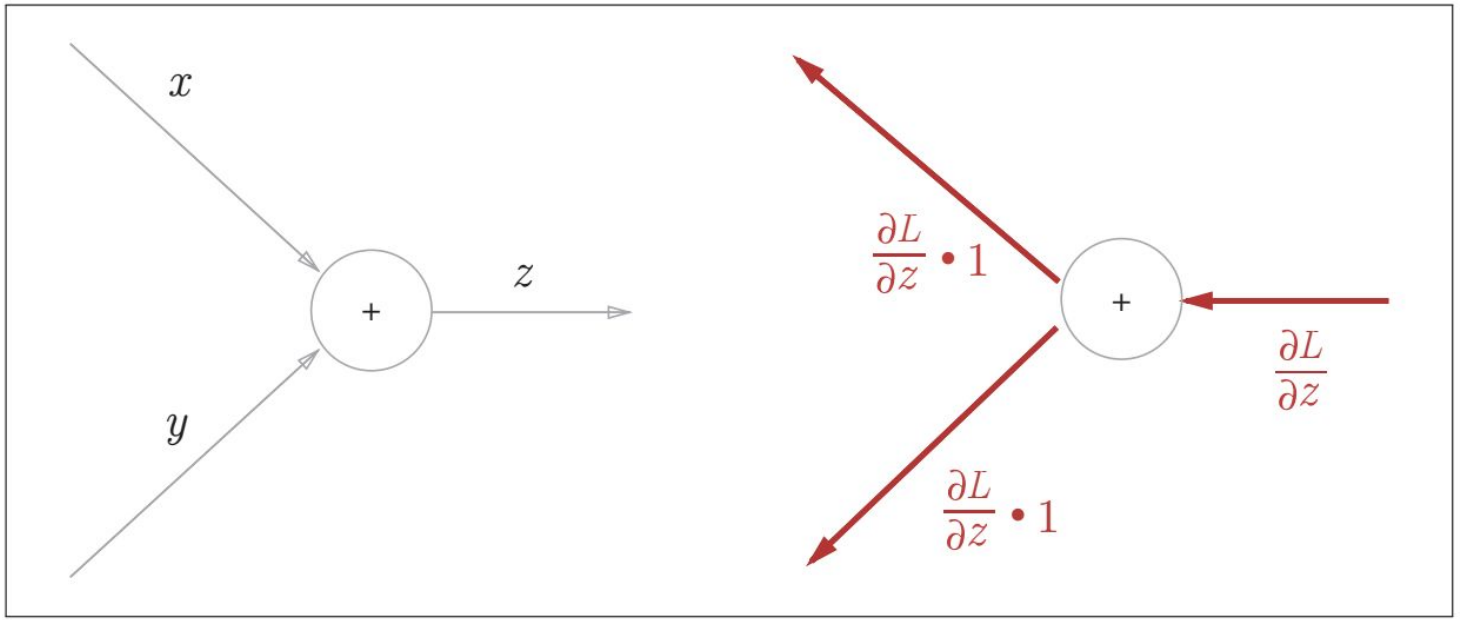


図5-9　加算ノードの逆伝播：左図が順伝播、右図が逆伝播。右図の逆伝播が示すように、加算ノードの逆伝播は、上流の値をそのまま下流へ流す

つまり、加算ノードの逆伝播は、入力信号をそのまま次のノードへ出力するだけ。

5.3.2 乗算ノードの逆伝播

z = xy という式について考えると、それぞれの偏微分はこうなる。

$$\frac{\partial z}{\partial x} = y$$
$$\frac{\partial z}{\partial y} = x \tag{5.6}$$

これは、順伝播の際の入力信号と反対の信号になっている。

計算グラフは次スライド

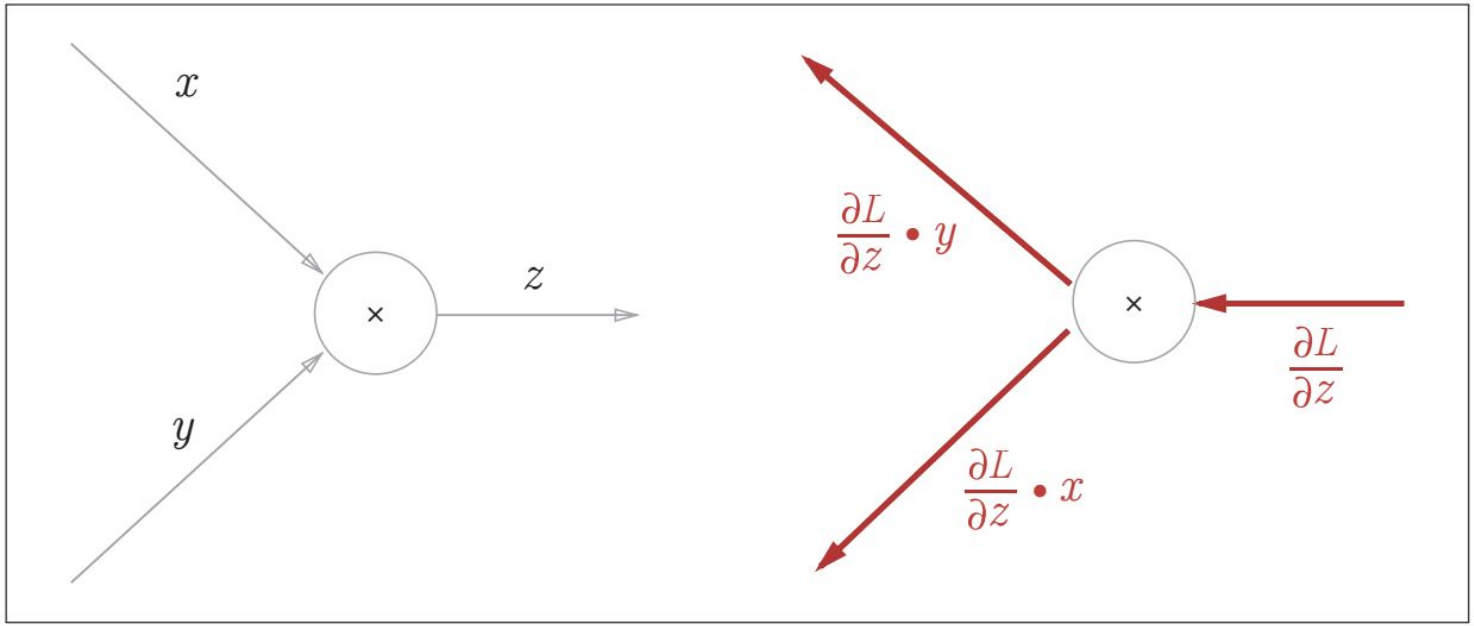
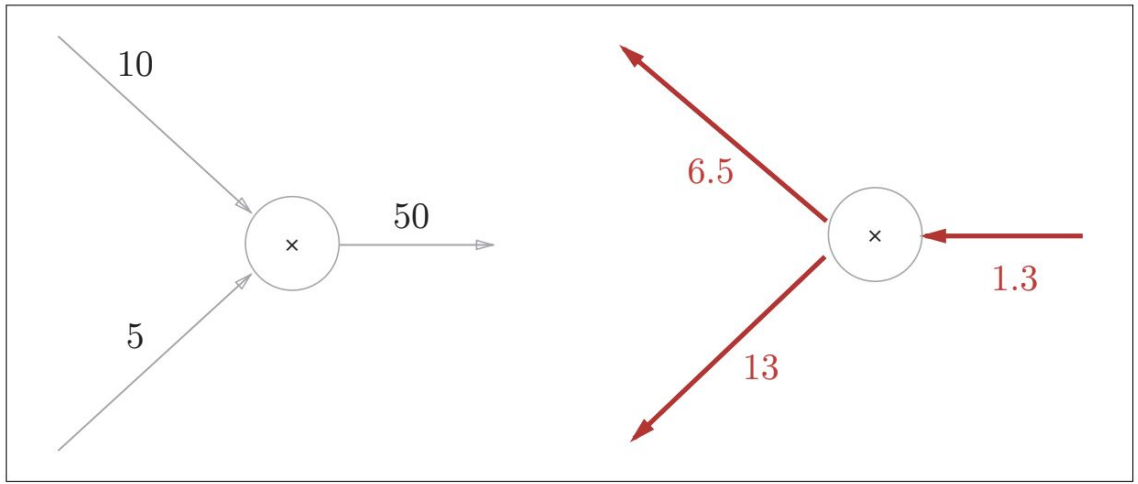


図 5-12　乗算の逆伝播：左図が順伝播、右図が逆伝播

入力が x のエッジには y 、y のエッジには x を乗算している

x = 5 , y = 10 , L = 1.3　のときは次のようになる。



1.3 * 5 = 6.5

1.3 * 10 = 13

図5-13　乗算ノードの逆伝播の具体例

乗算の逆伝播には、順伝播の際の入力信号の値が必要になるので、実装の際には順伝播の入力信号を保持しておく必要があります。

5.3.3 りんごの例で考えてみる

りんごを買うときの支払金額は、りんごの値段、りんごの個数、消費税、の 3つが関係します。これら3つの変数が、支払い金額にどれだけ影響するか、という問題を解くことになります。

りんごの値段が1 (円)増えたとき、支払い金額は 2.2 だけ増える

りんごの個数が1 (個)増えたとき、支払い金額は 110 だけ増える

消費税が1 (100%)増えたとき、支払い金額は 200 増える

と言ったことを右の図から読み取れます。

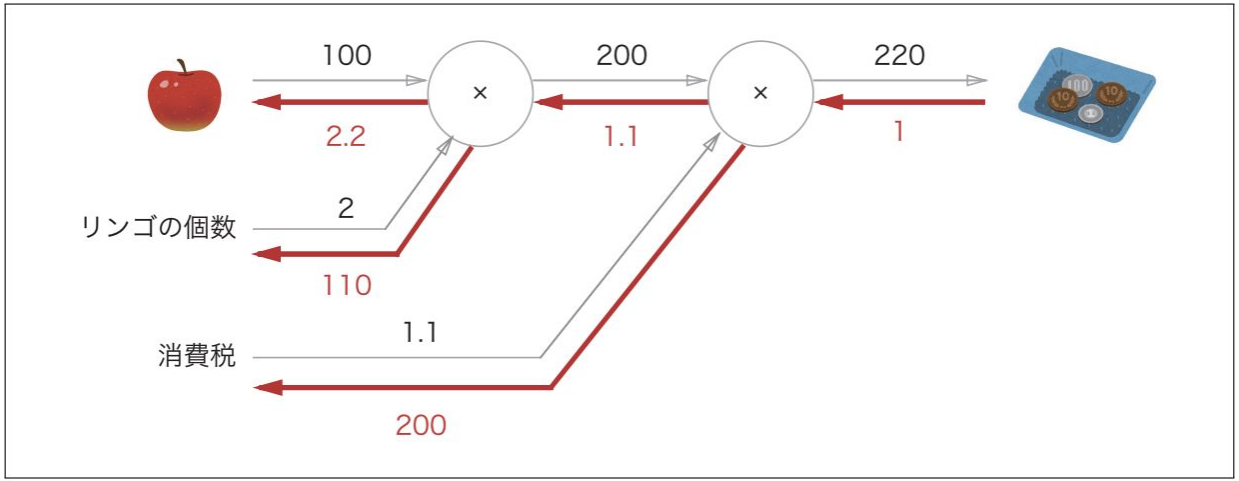


図5-14　リンゴの買い物の逆伝播の例

教科書に例題があるので解きたい人は解いてみてね

答えは教科書の140pです

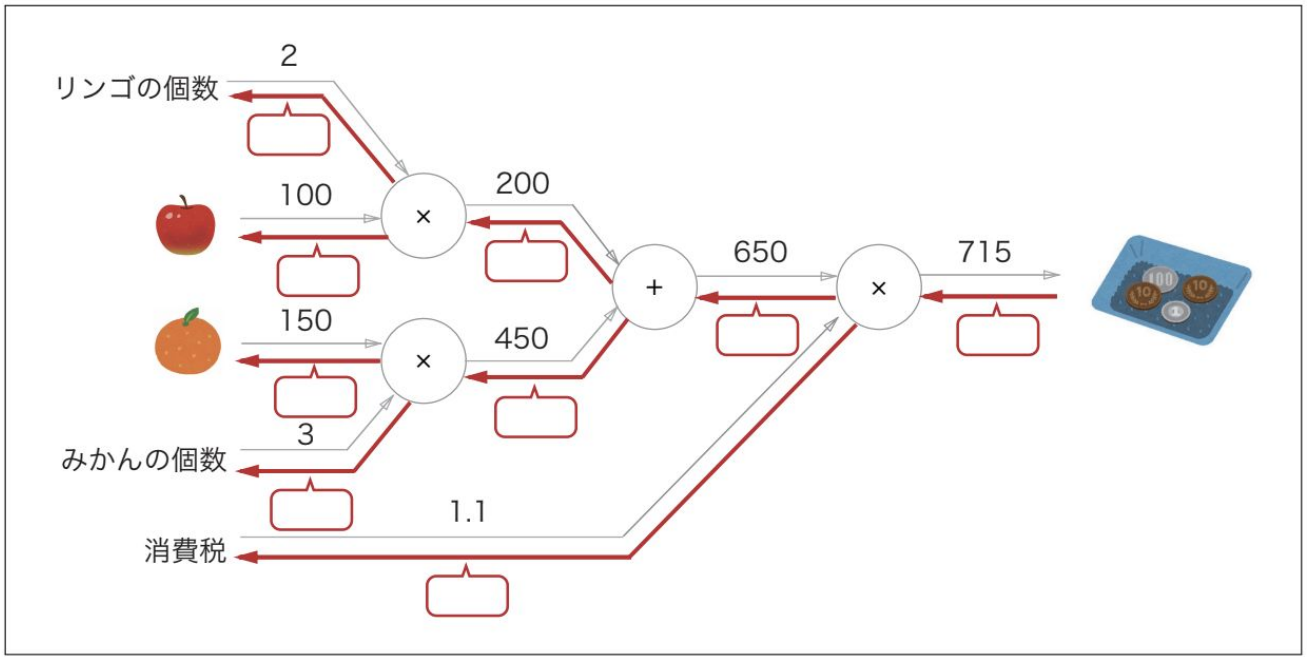


図5-15　リンゴとみかんの買い物の逆伝播の例：四角に数字を入れて逆伝播を完成させよう

5.4 単純なレイヤの実装

5.4.1 乗算レイヤの実装

レイヤの実装には、順伝播 (forward())と逆伝播(backward())を実装します。

forward() では、入力した2つの値の乗算を返しています。

backward() では、入力は上流から伝わってきた微分 (dout)で、微分に順伝播の入力信号を反対にした値を乗算して返しています。

__init__() は、順伝播時の x と y の入力値を保存するためにあります。

```
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 # x と y をひっくり返す
        dy = dout * self.x

        return dx, dy
```

先ほどのコードを使ってりんごの買い物を実装すると右のようになる。

この時、順伝播(forward)と逆伝播(backward)で関数を呼び出す順番が逆になることに注意。

また、backward()の引数は、「順伝播の際の出力変数に対する微分」を入力しています。

順伝播の mul_apple_layer は、
入力：apple, apple_num
出力：apple_price

逆伝播の mul_apple_layer は、
入力：dapple_price
出力：dapple, dapple_num

buy_apple.py

```
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))  # 220
print("dApple:", dapple)  # 2.2
print("dApple_num:", int(dapple_num))  # 110
print("dTax:", dtax)  # 200
```

5.4.2 加算レイヤの実装

加算レイヤでは、初期化は特に必要ないので _init_() は pass

forward()では、2つの入力を加算してそのまま出力

backward()では、上流から伝わってきた値の微分 (dout)をそのまま下流に流しています。

```
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
```

スライド4つ前の問題を実装すると右のようになる。

(1) ~ (4) はレイヤの順番。準伝播と逆伝播で順番が逆になるようにする。

```
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)
```

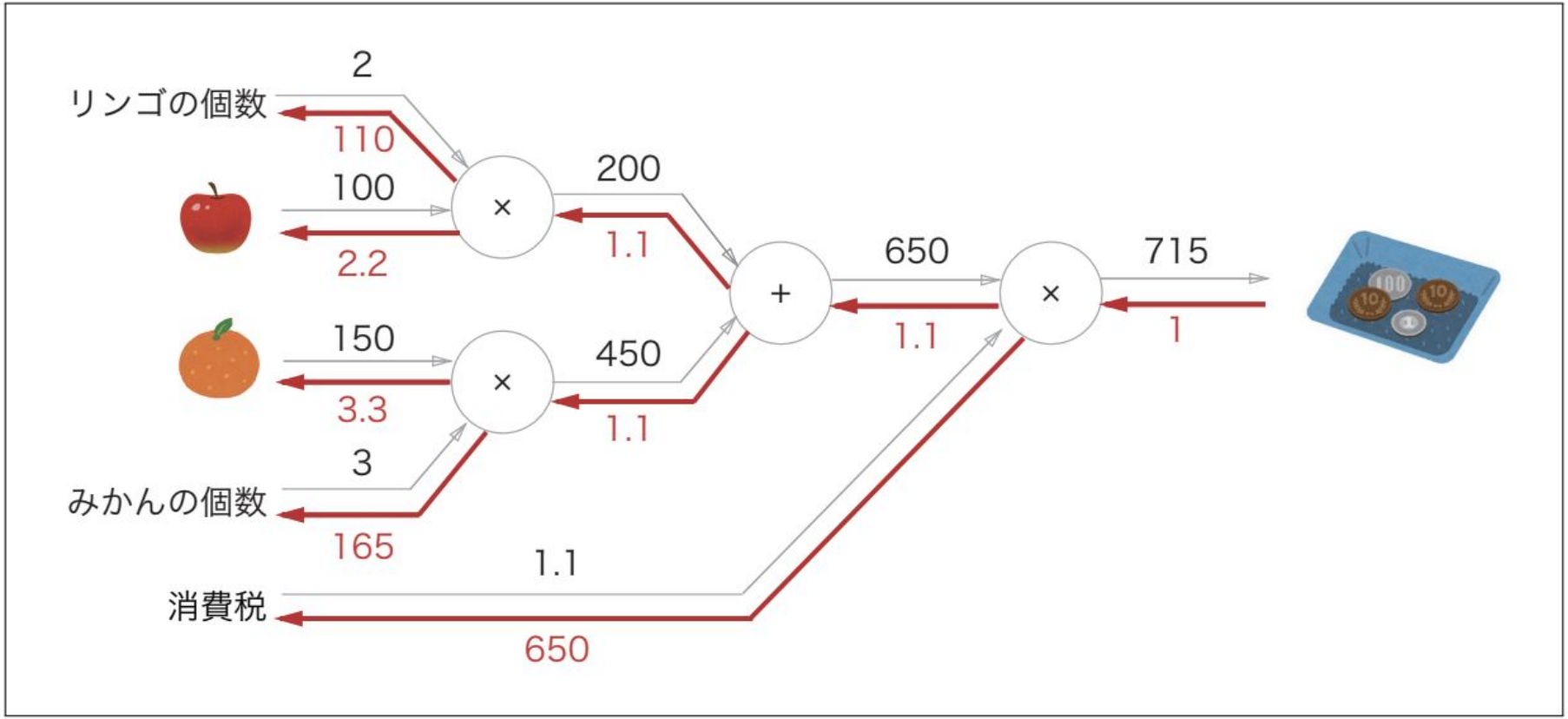


図5-17　リンゴ2個とみかん3個の買い物

5.5 活性化関数レイヤ

活性化関数・・・閾値を境にして出力が切り替わる関数で、入力信号の総和がどのように活性化するかを表す。

ここで実装する活性化関数

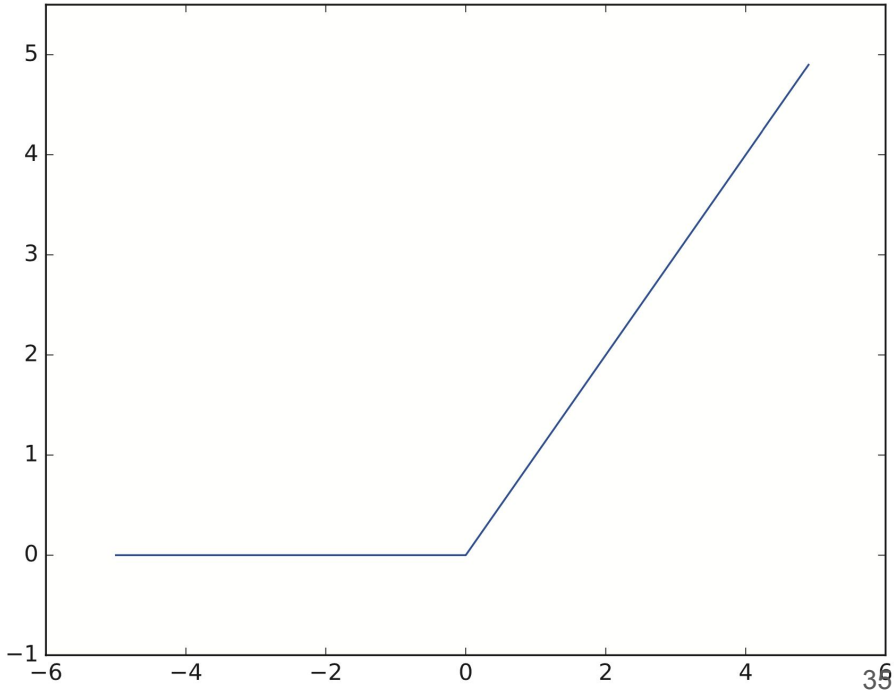
・ReLU関数

・Sigmoid関数

5.5.1 ReLU関数

ReLU関数は、入力が0を超えていれば入力をそのまま出力し、0以下ならば0を出力する関数である。

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



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

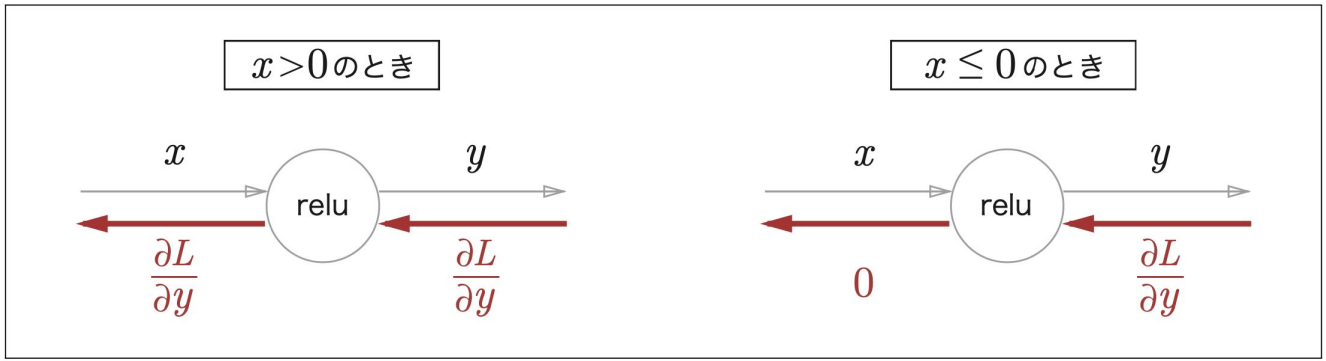
xに関するyの微分

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

上流から伝達された値を $\frac{\partial L}{\partial y}$ とすると、

$x > 0$ のとき, $\frac{\partial L}{\partial y} \times 1 = \frac{\partial L}{\partial y}$

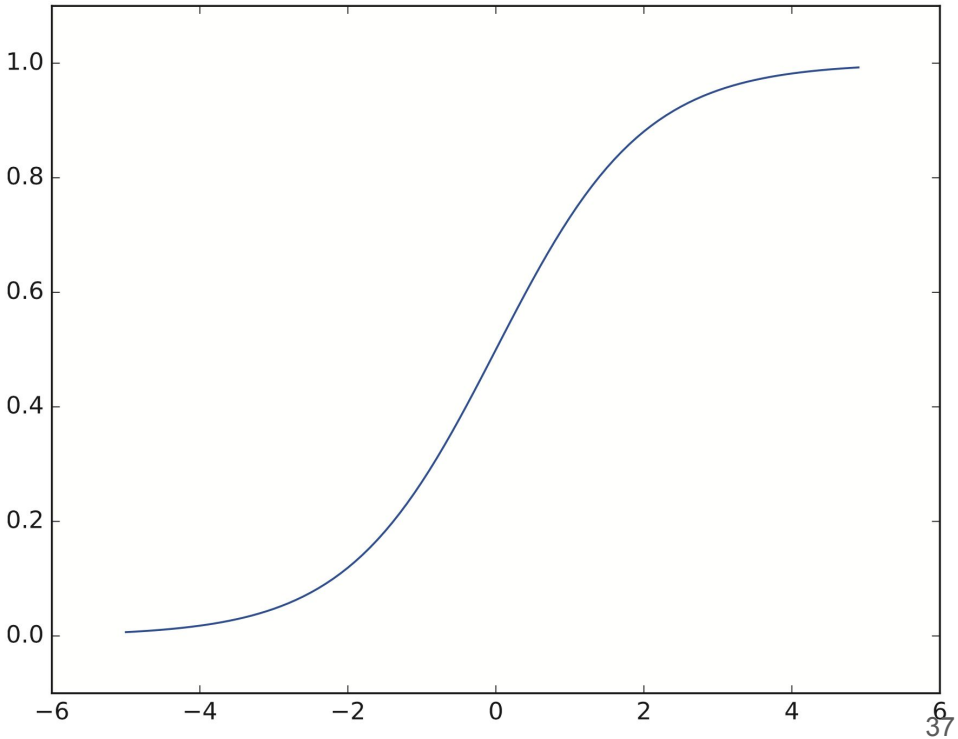
$x \leq 0$ のとき, $\frac{\partial L}{\partial y} \times 0 = 0$



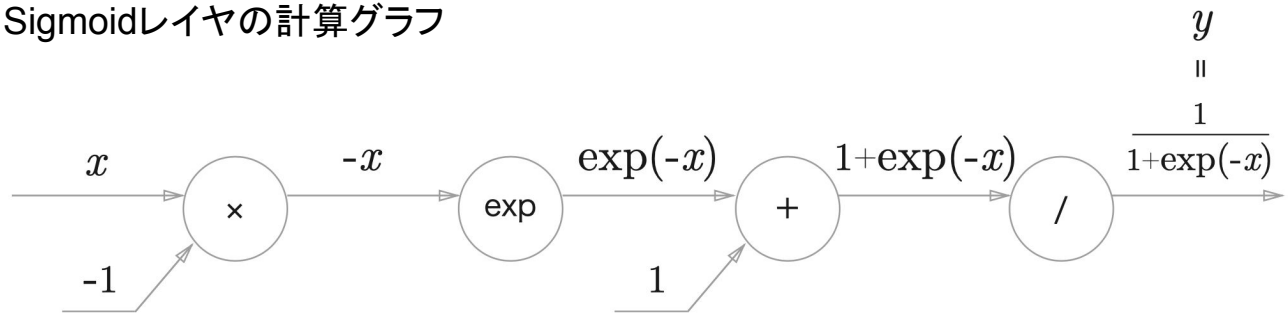
5.5.2 Sigmoid関数

Sigmoid関数は、入力値を0.0から1.0までの値に変換する関数。

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



Sigmoidレイヤの計算グラフ



×，＋ノードの他にexpノードは y = exp(x)，/ノードは y = 1/xの計算を行う。

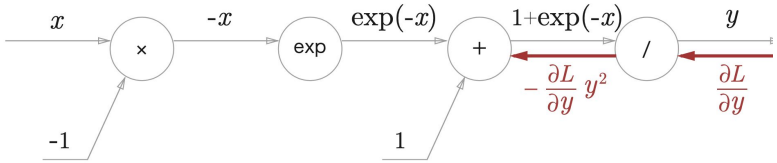
Sigmoid関数における逆伝播の様子

STEP1

/ノードは y = 1/x を表すから微分すると、

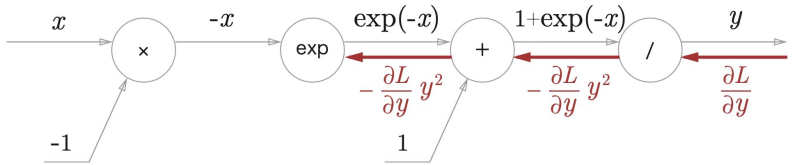
$$\frac{\partial y}{\partial x} = -\frac{1}{x^2}$$
$$= -y^2$$

微分値を上流の値と乗算して下流に伝播する。



STEP2

＋ノードは上流の値を下流にそのまま流す

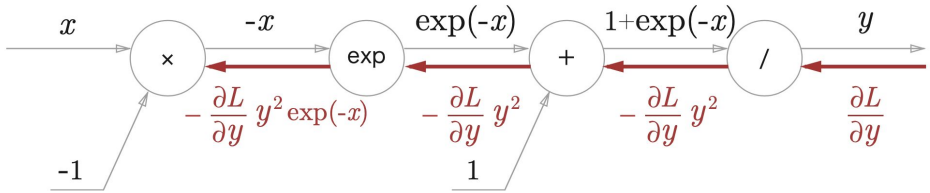


STEP3

expノードは y = exp(x)で、微分式は以下

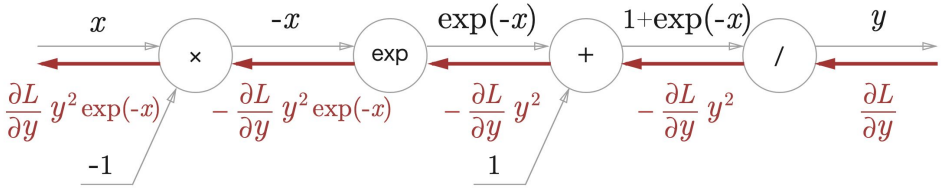
$$\frac{\partial y}{\partial x} = \exp(x)$$

微分値を上流の値と乗算して下流に伝播する。

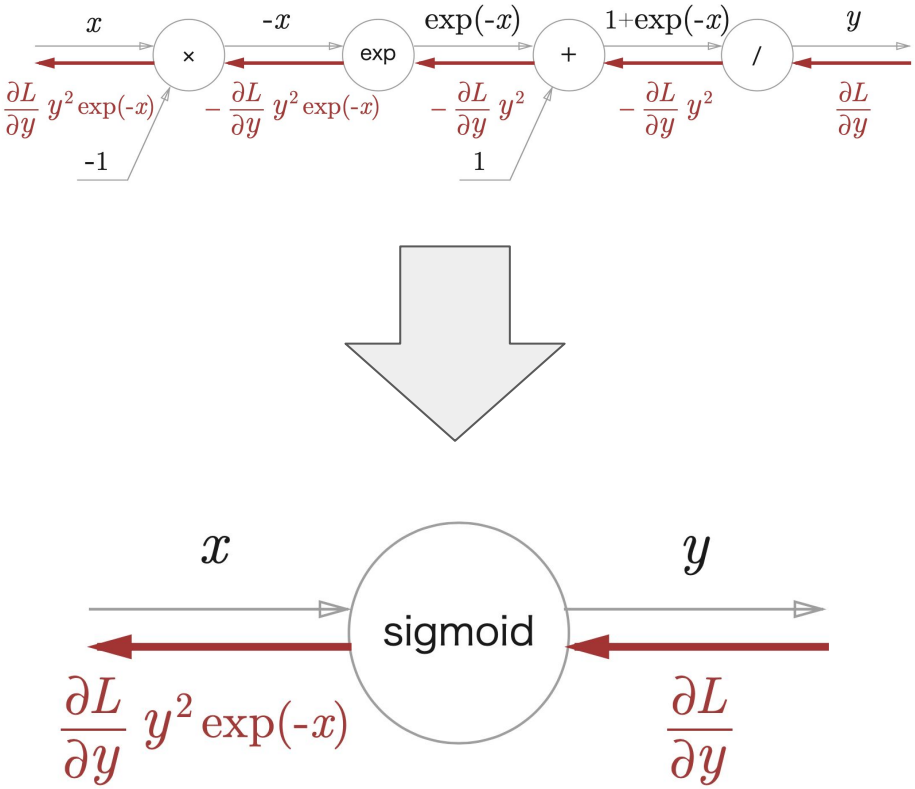


STEP4

×ノードは順伝搬時の値をひっくり返して乗算する。（ここでは-1を乗算する）



ここでSTEP4のSigmoidレイヤの計算グラフに注目すると、逆伝播で下流に伝達する値は順伝播の入力と出力であるxとyのみで計算できることがわかる。
よって、右下図のようにグループ化した Sigmoidノードとして書くことができる。

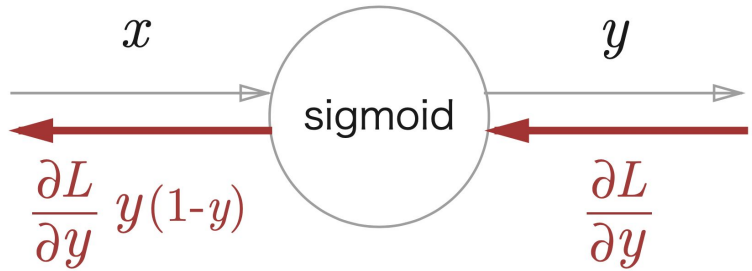


簡略化するメリット
・逆伝播の際の途中の計算を省略できる
・細かい中身を気にすることなく、入力と出力だけに注目できる

ここで下流に伝達する値を右の計算式のように変換する。変換することによって右下図の計算グラフでSigmoidレイヤを表すことができる。

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

変換前は入力と出力が必要であったのに対し、変換後は順伝播時の出力である yのみで逆伝播の計算ができる。



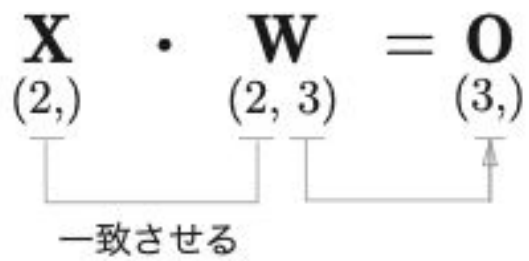
5.6 Affine / Softmax レイヤの実装

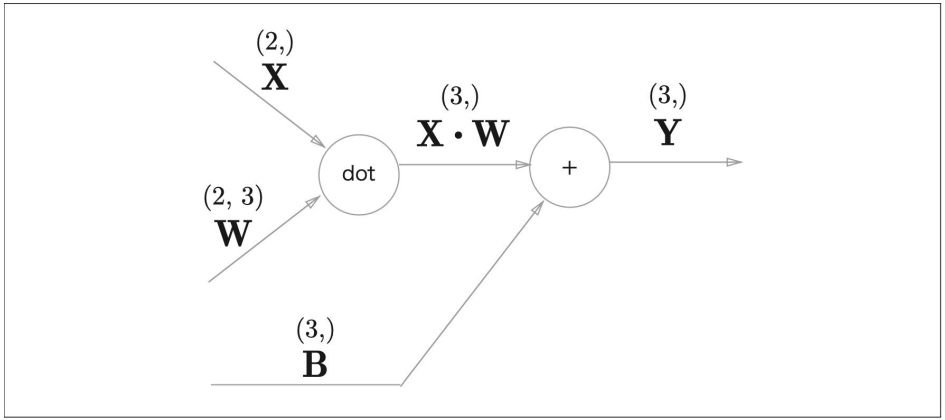
NNの順伝播では、重み付き総和を計算するために、行列の内積を用いる

Y = np.dot(X,W) + b

その際に注意することとして、対応する**次元数を一致**させることが需要

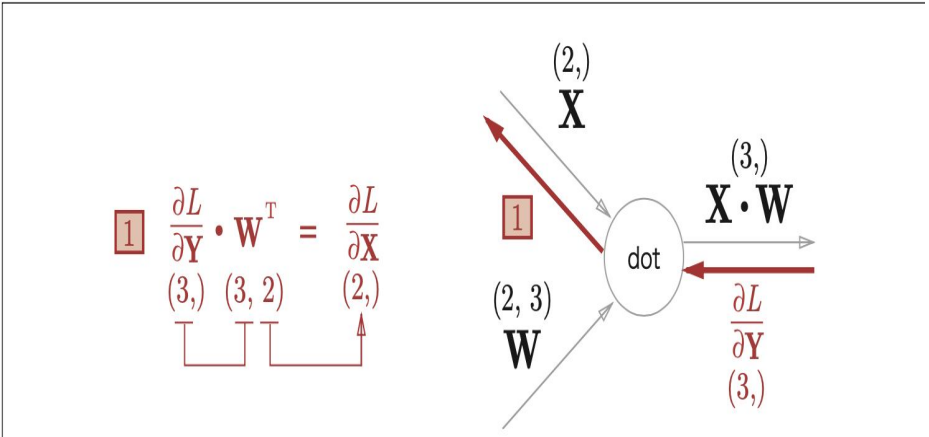
NNの順伝播で行う内積の処理するレイヤを**Affineレイヤ**と呼ぶ

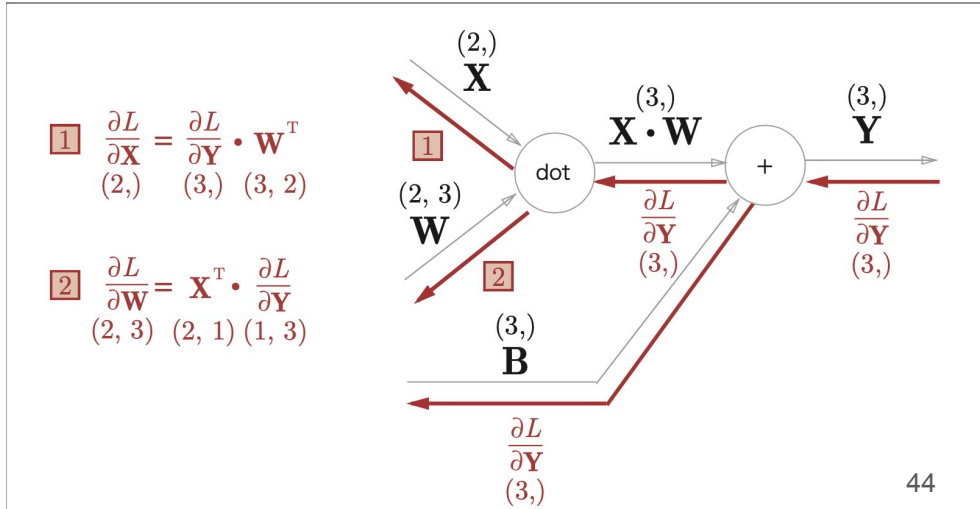




5.6.1 Affineレイヤ

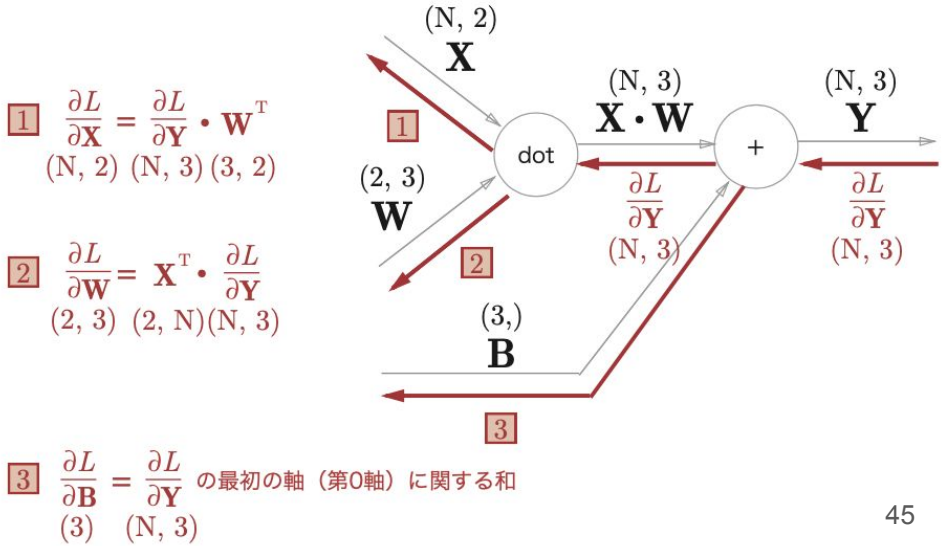
Affineレイヤの逆伝播は、下図のようになる
各変数の形状に注意しなければならない(XとdI/dxやWとdI/dwは同じ形状)
形状をに注意しないといけないのは、行列の内積では対応する**次元数を一致**させなければならない





5.6.2 バッチ版Affineレイヤ

N個のデータをまとめて順伝播する場合、つまり、バッチ版Affineレイヤを考える
先ほどの例と異なる点として、入力Xの形状(N,2)となる
逆伝播の際に、行列の形状に注意する



5.6.2 バッチ版Affineレイヤ

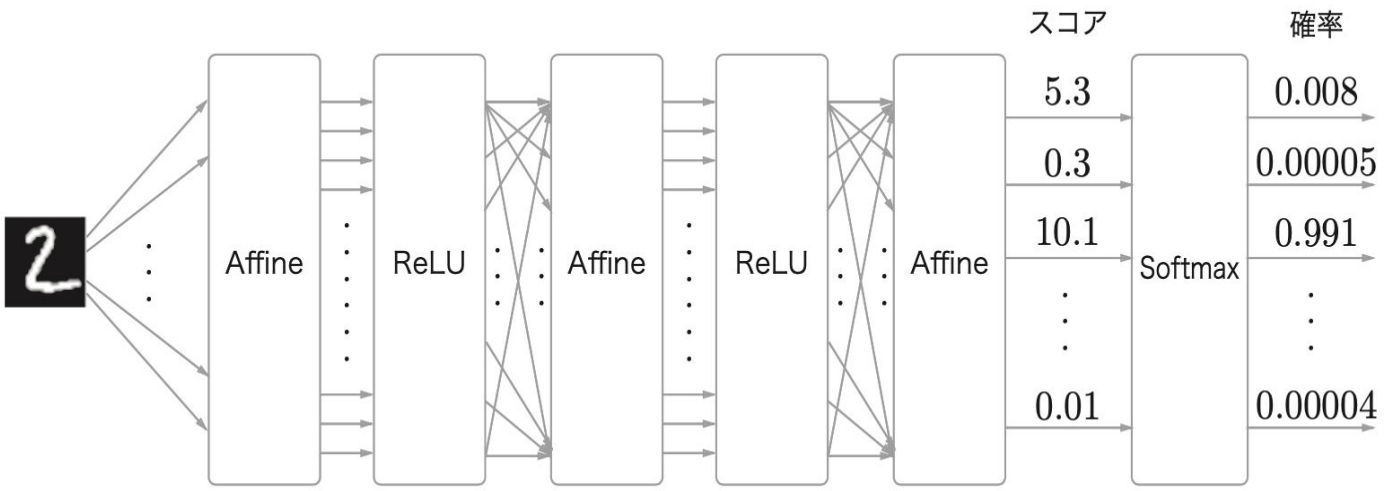
さらに、注意することとして、加算のところである
逆伝播では、それぞれのデータの逆伝播の値がバイアスの要素に集約される必要がある

```
>>> X_dot_W = np.array([[0, 0, 0], [10, 10, 10]])
>>> B = np.array([1, 2, 3])
>>>
>>> X_dot_W
array([[ 0,  0,  0],
       [ 10, 10, 10]])
>>> X_dot_W + B
array([[ 1,  2,  3],
       [11, 12, 13]])
```

```
>>> dY = np.array([[1, 2, 3,], [4, 5, 6]])
>>> dY
array([[1, 2, 3],
       [4, 5, 6]])
>>>
>>> dB = np.sum(dY, axis=0)
>>> dB
array([5, 7, 9])
```

5.6.3 Softmax-with-Lossレイヤ

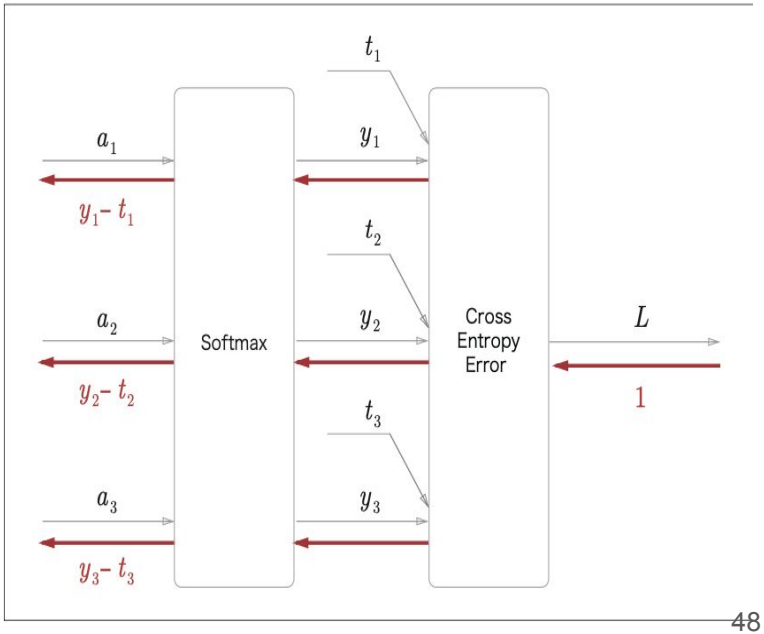
最後は、出力層を見ていく
Softmax関数は、入力された値を正規化(出力の値の総和を1)する



5.6.3 Softmax-with-Lossレイヤ

そして、損失関数(交差エントロピー誤差)を含めた**Softmax-with-Lossレイヤ**を見ていく

入力(a1,a2,a3)
Softmax関数後(y1,y2,y3)
教師ラベル(t1,t2,t3)
データの損失(L)
Softmax関数の逆伝播
　　(y1-t1,y2-t2,y3-t3)



5.6.3 Softmax-with-Lossレイヤ

NNの学習の目的として、出力値を教師ラベルに近づけるようにパラメータの調整することである
そのため、誤差を効率良く前レイヤに伝える必要がある

具体的な例として、**教師ラベル**が(0,1,0)とする
1,softmaxの出力(0.3,0.2,0.5)　正解ラベルに対する確率20%
　逆伝播(0.3,-0.8,0.5)　大きな誤差から大きな内容を学習する

2,softmaxの出力(0.01,0.99,0)　正解ラベルに対する確率99%
　逆伝播(0.01,-0.01,0) 小さな誤差のため学習する内容も小さい

5.7 誤差逆伝播法の実装

前節までに実装したレイヤを組み合わせることにより、ニューラルネットワークを構築することができる。

ニューラルネットワークの要素として**重み**と**バイアス**がある。これらを訓練データに適応するように調整することを**学習**と呼ぶ。

学習には大きく分けて4つの手順が存在する。

学習の手順

1. ミニバッチ
    a. 訓練データの中からランダムに一部のデータを選び出す
2. 勾配の算出
    a. 各重みパラメータに関する損失関数の勾配を求める
3. パラメータの更新
    a. 重みパラメータを勾配方向に微小量だけ更新する
4. 繰り返す
    a. 1,2,3を繰り返す

誤差逆伝播法が登場するのは勾配の算出の場面である

レイヤをOrderdDictという順番付きのディクショナリで保存している

レイヤを正しい順番で連結し、呼び出すことで内部でAffineレイヤやReLUレイヤが順伝播と逆伝播を正しく処理してくれる

ネットワークそのものが層が深くなり大きくなったとしても、レイヤを追加するだけで構成できるようになる

コード全体ははtwo_layer_net.pyで保存されているので参照してほしい

```
        # 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()

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

    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)
```

誤差逆伝播法の勾配確認

勾配確認とは

数値微分で求めた結果と、誤差逆伝播法で求めた勾配が一致すること(ほとんど近い値にあること)を確認する作業のこと

数値微分は誤差逆伝播法の正しさを確認する場面で実践的に必要とされる

5.8 まとめ(学んだこと)

- 計算グラフを用いれば、計算過程を視覚的に把握することができる
- 計算グラフのノードは局所的な計算によって構成される。局所的な計算が全体の計算を構成する
- 計算グラフの順伝播は、通常の計算を行う。一方、計算グラフの逆伝播によって、各ノードの微分を求めることができる
- ニューラルネットワークの構成要素をレイヤとして実装することで、勾配の計算を効率的に求めることができる（誤差逆伝播法）
- 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）