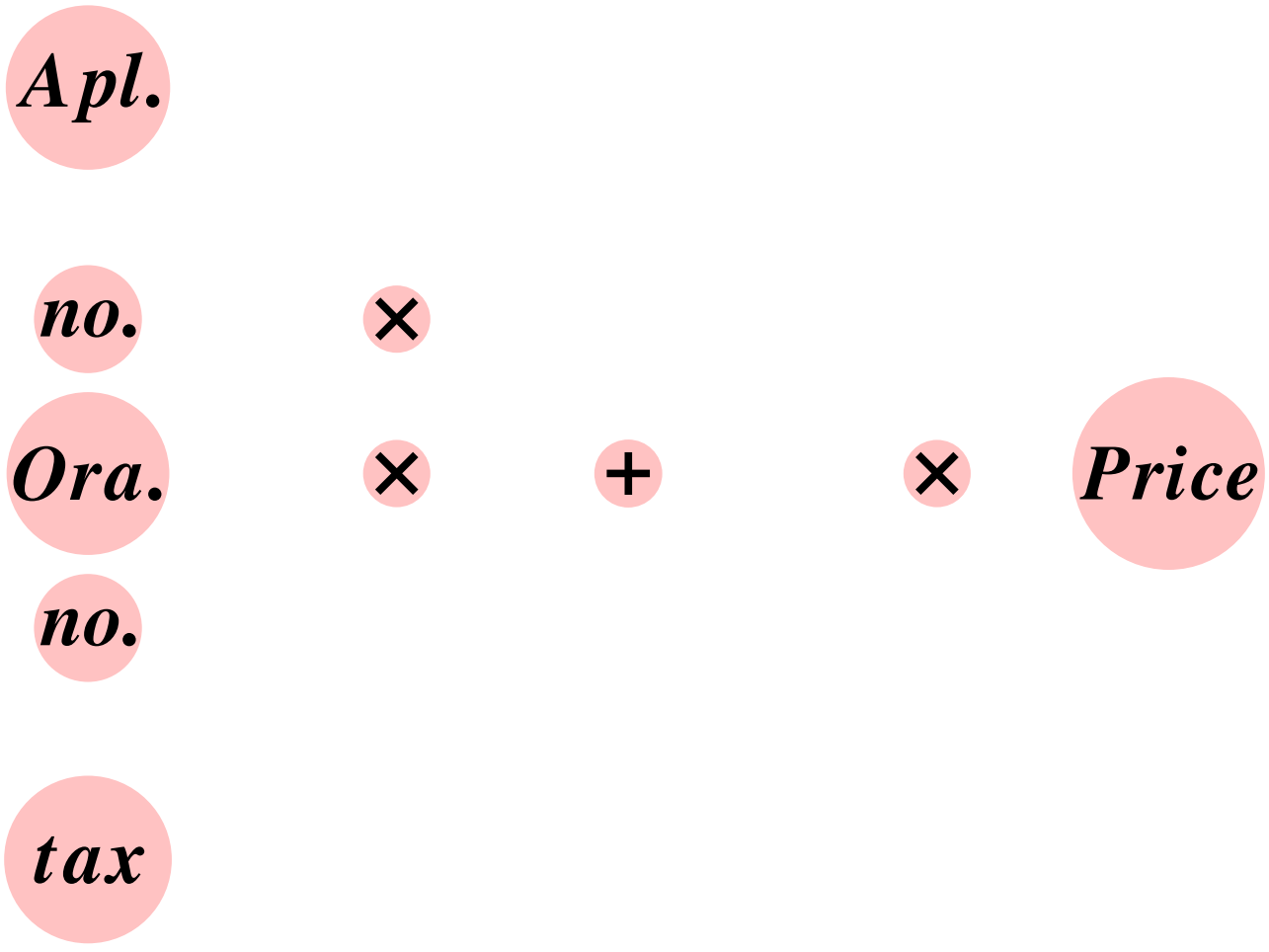
バックプロパゲーション【逆誤差伝播法】(2)

前田利之（まえだ　としゆき）
aipr@e-chan.jp

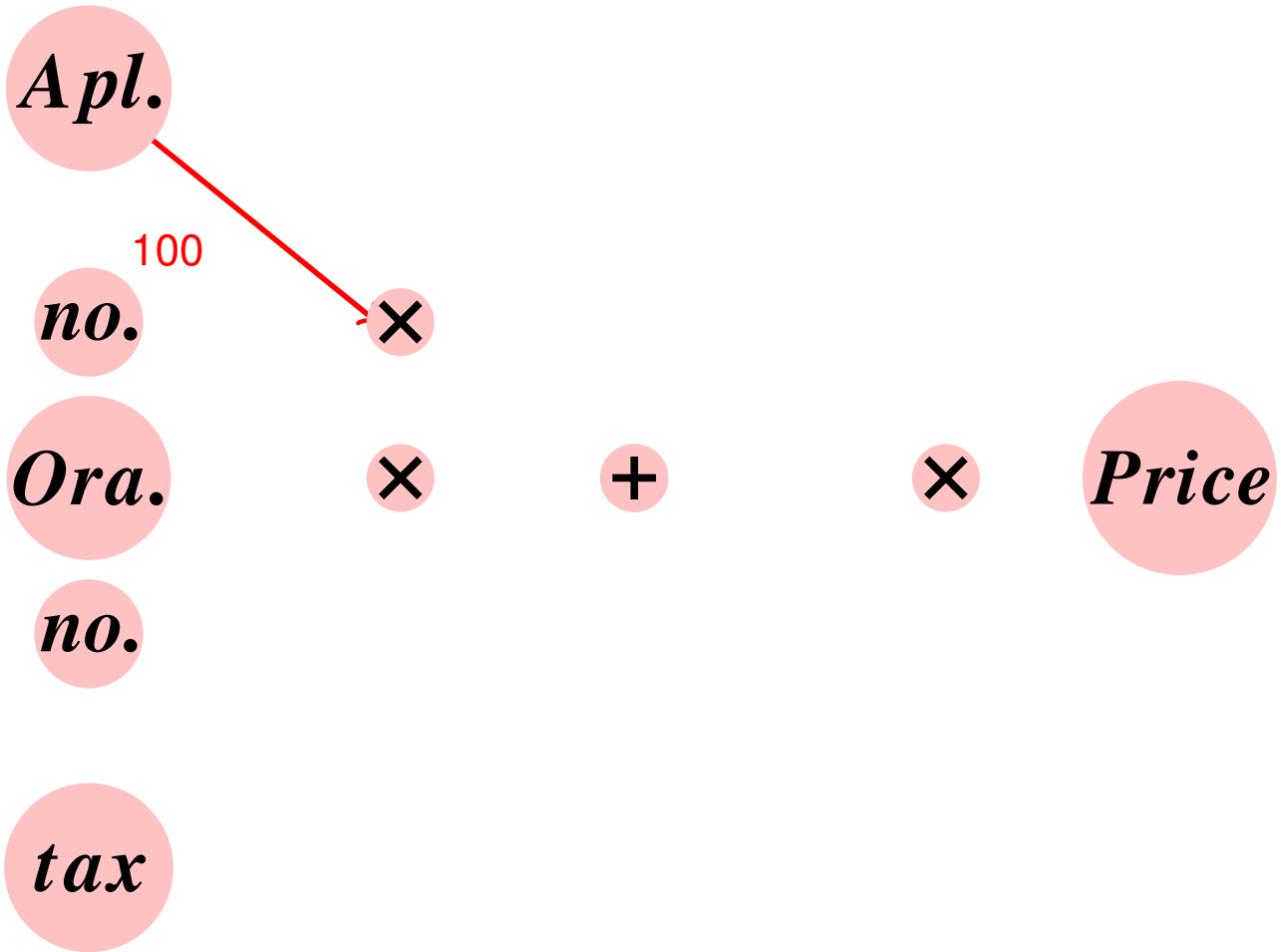
阪南大学　経営情報学部

(2024.11.22)

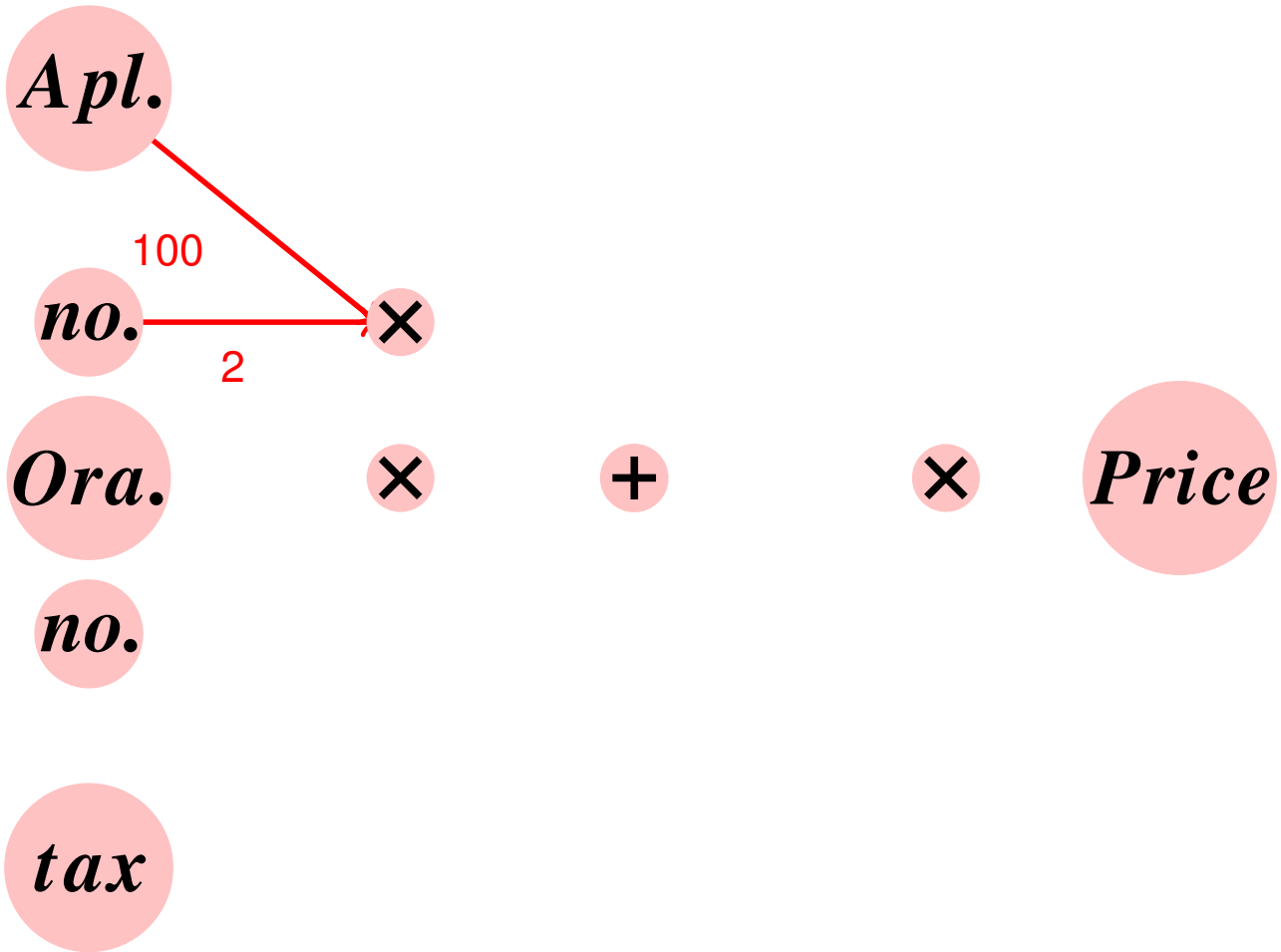
復習をかねて：リンゴとミカンの計算の逆伝搬の例



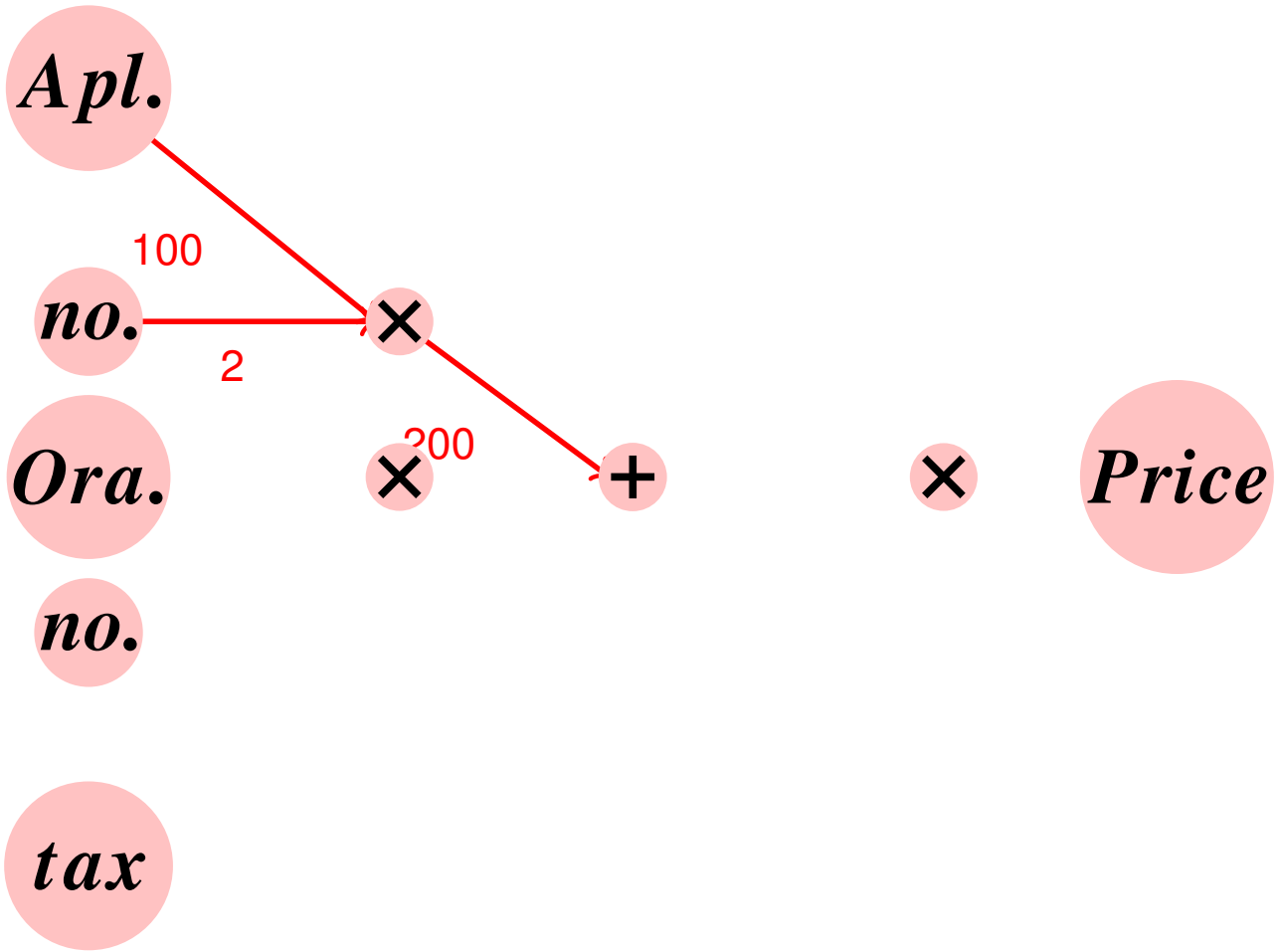
復習をかねて：リンゴとミカンの計算の逆伝搬の例



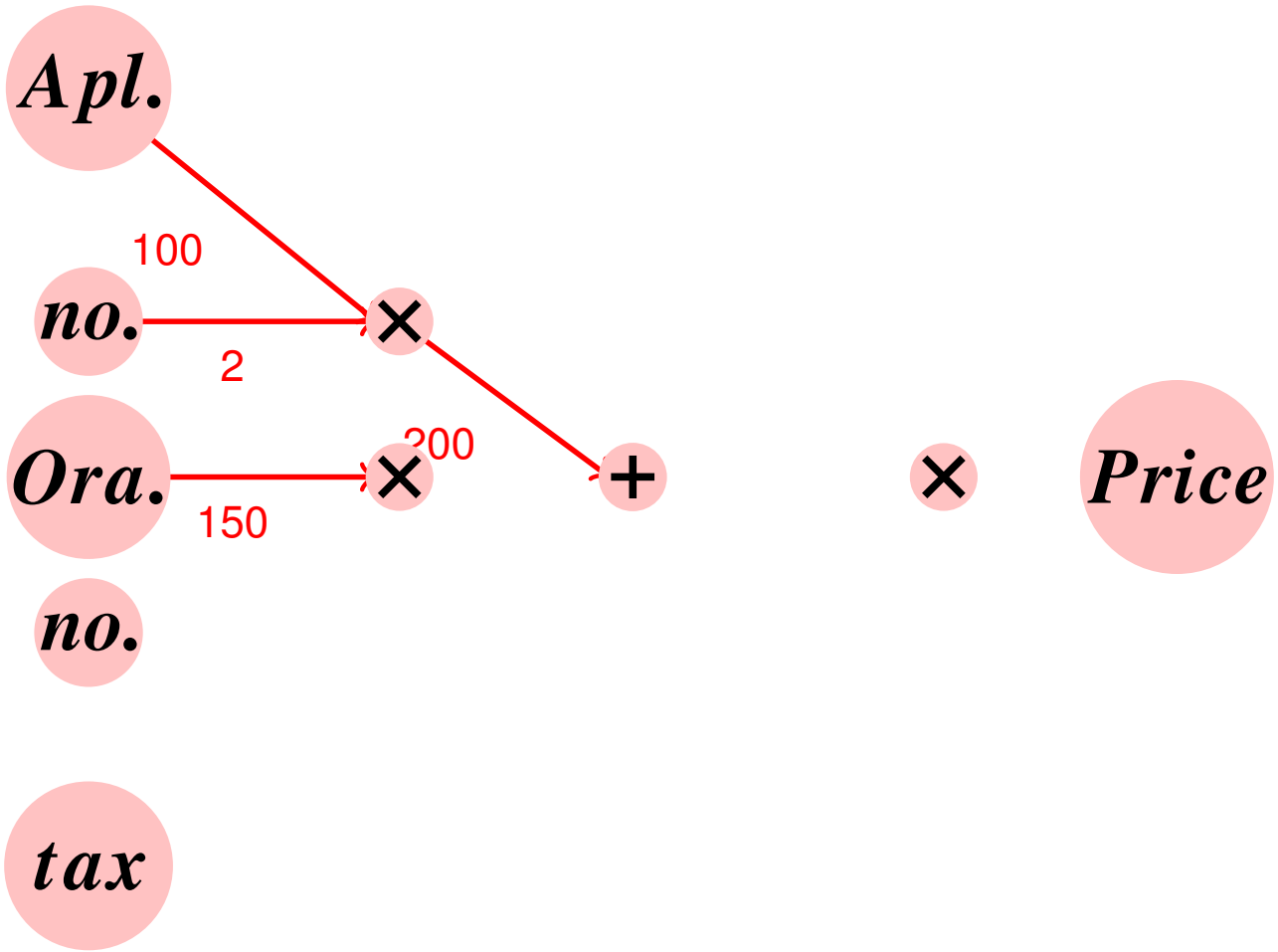
復習をかねて：リンゴとミカンの計算の逆伝搬の例



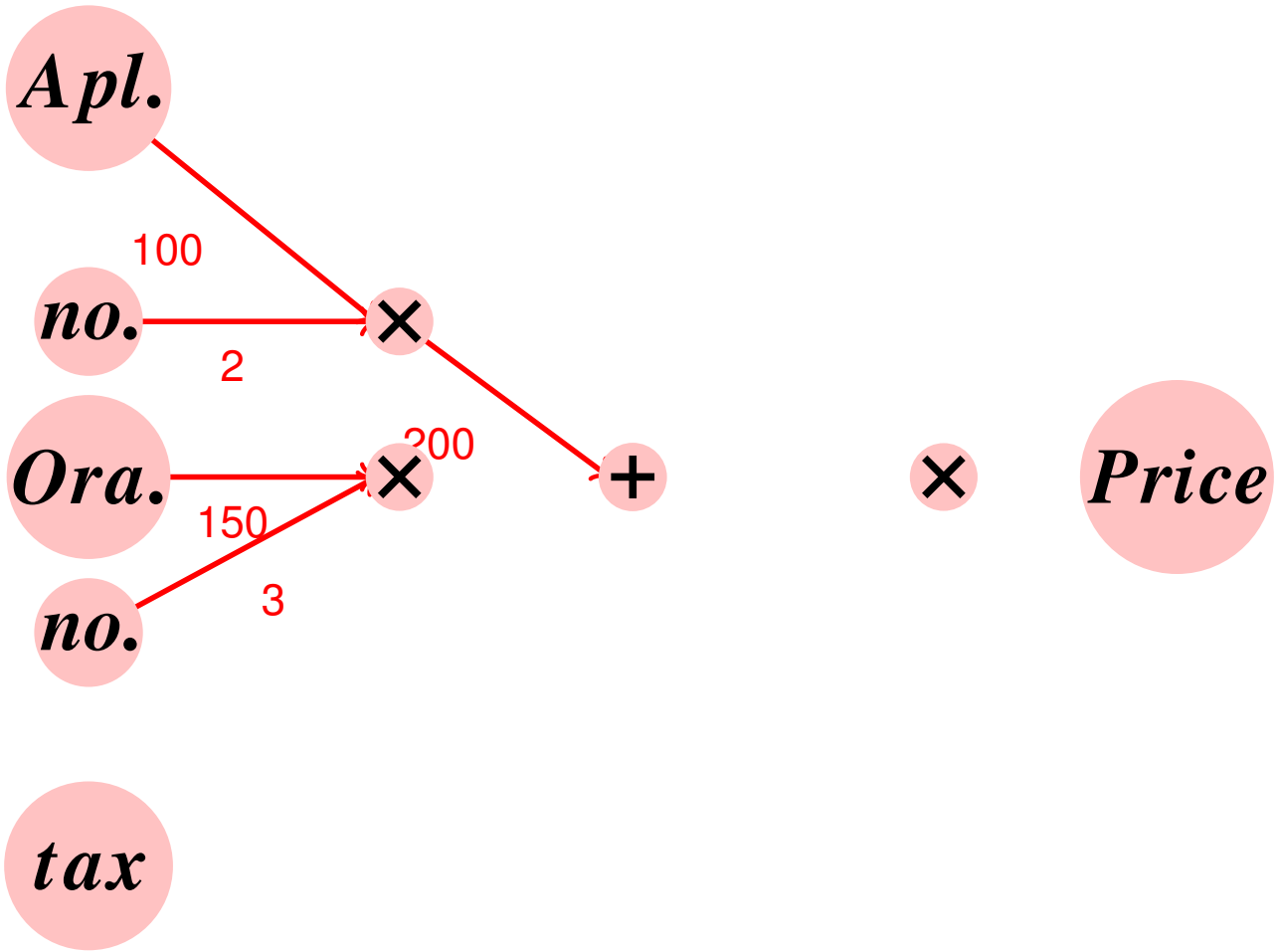
復習をかねて：リンゴとミカンの計算の逆伝搬の例



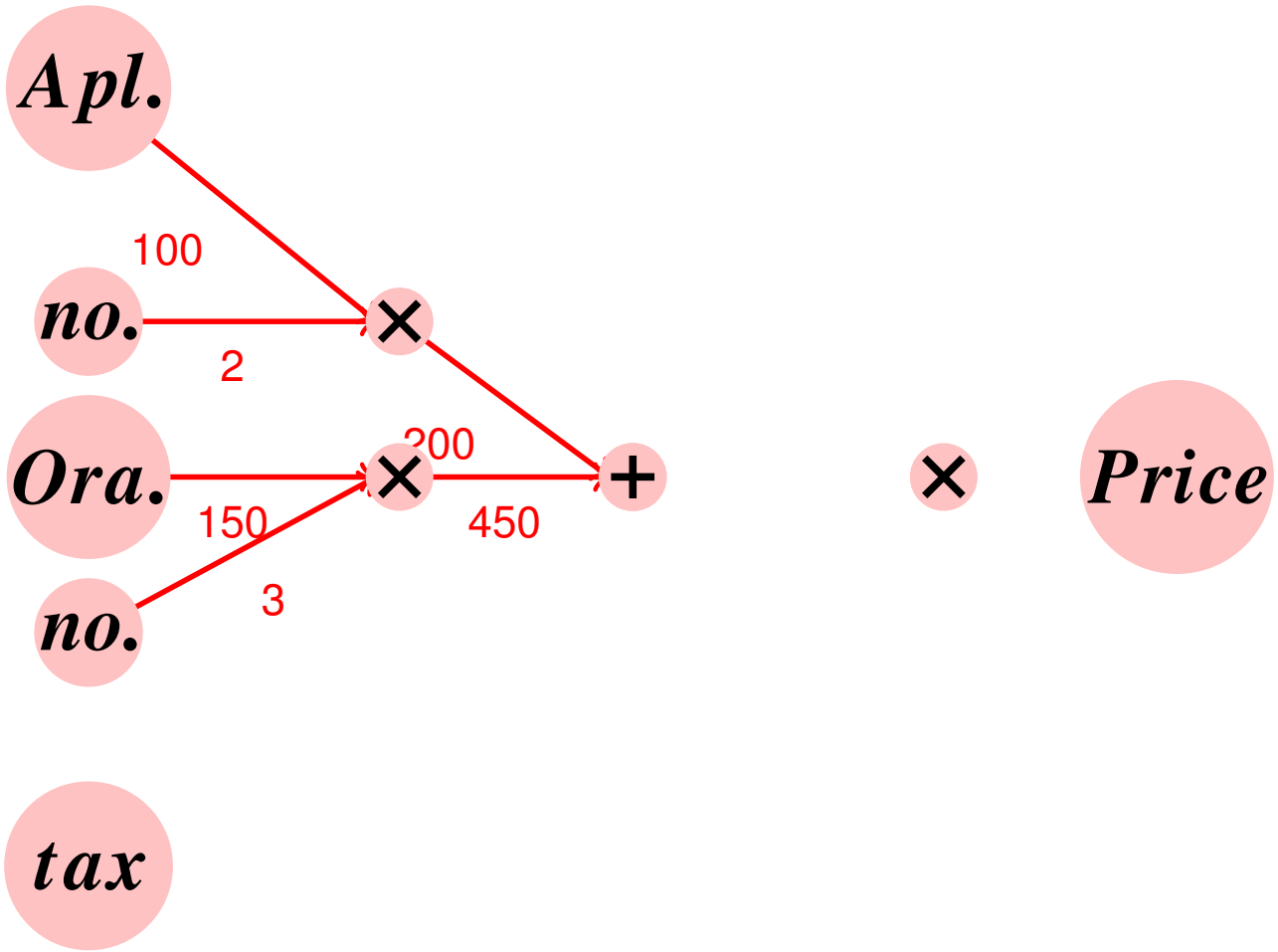
復習をかねて：リンゴとミカンの計算の逆伝搬の例



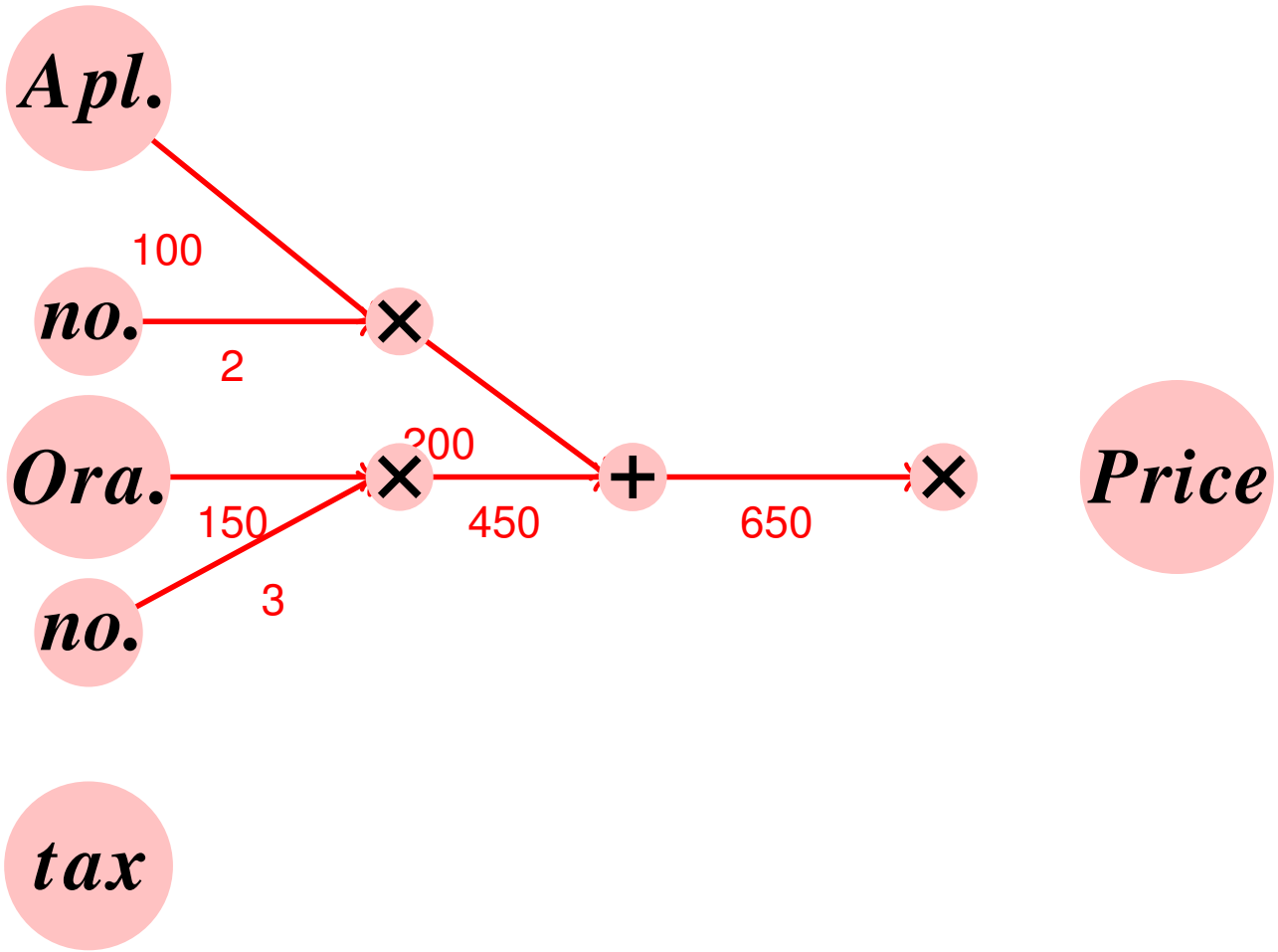
復習をかねて：リンゴとミカンの計算の逆伝搬の例



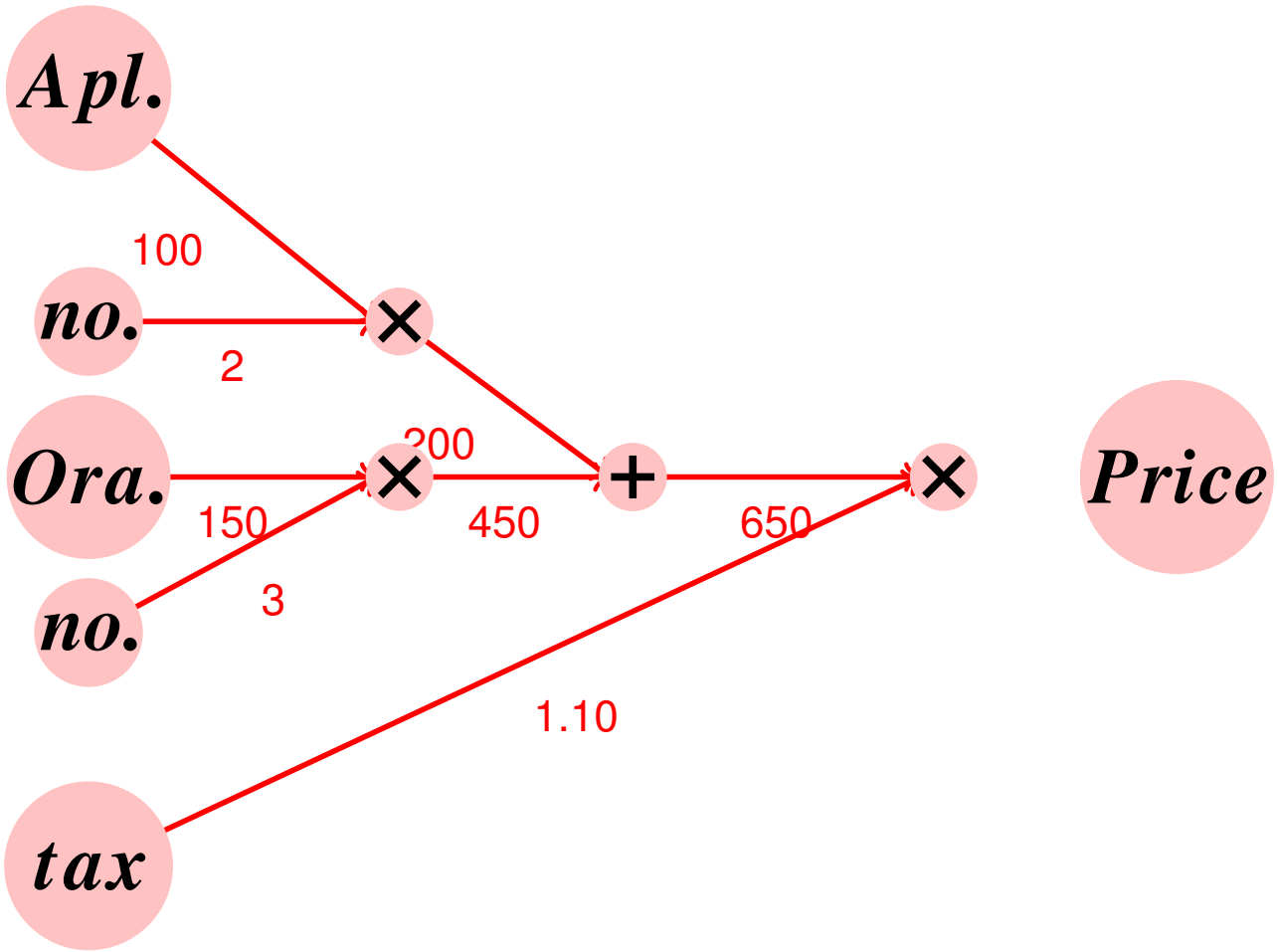
復習をかねて：リンゴとミカンの計算の逆伝搬の例



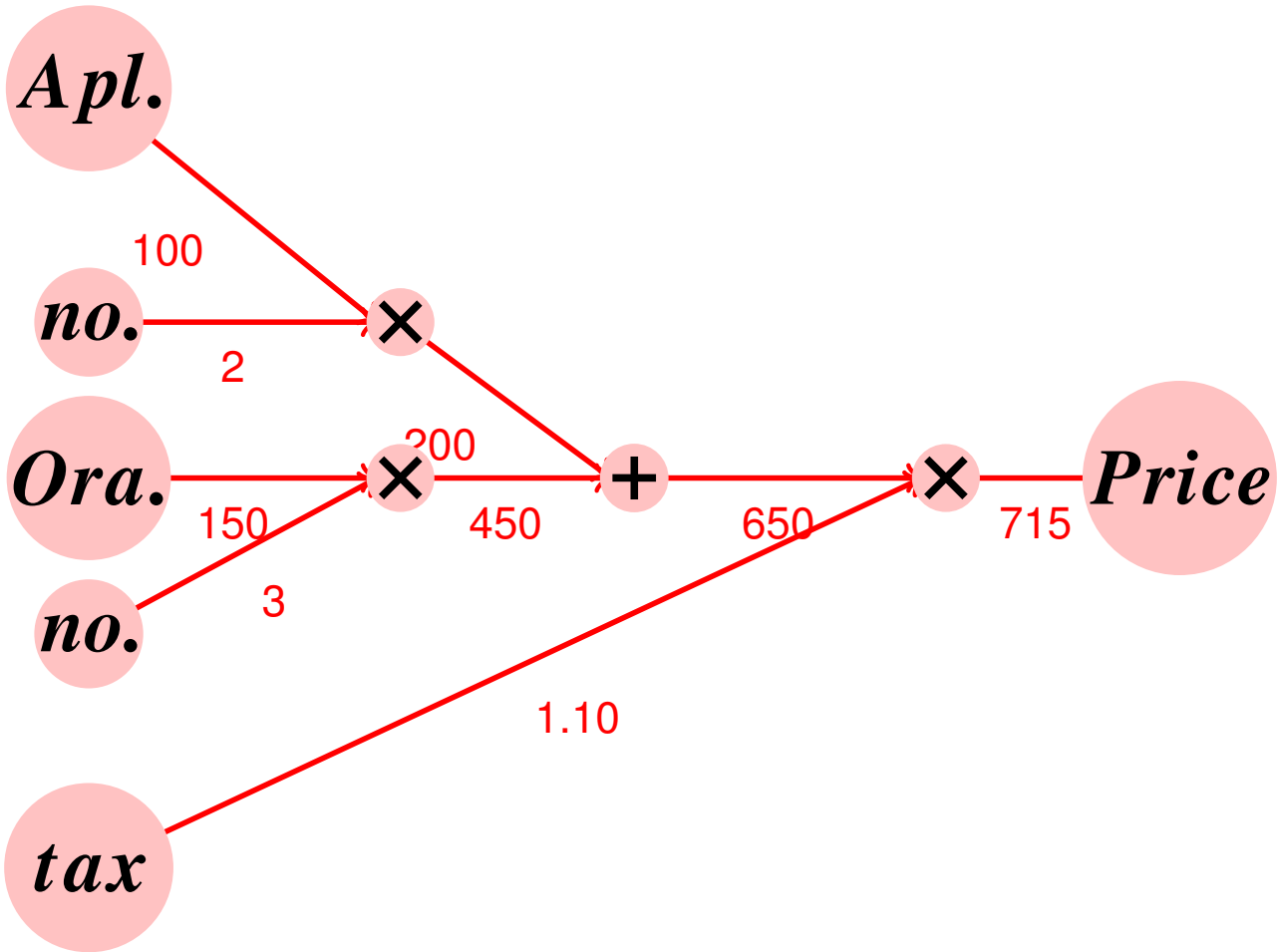
復習をかねて：リンゴとミカンの計算の逆伝搬の例



復習をかねて：リンゴとミカンの計算の逆伝搬の例



復習をかねて：リンゴとミカンの計算の逆伝搬の例



復習：加算・乗算レイヤの実装

☆以下によりリンゴとミカンの計算の逆伝搬を実行する（ buy_a_o.py のリンク先を Colab. にコピー＆ペーストし実行する）

```
# coding: utf-8
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
import numpy as np
sys.path.append(bd)
apple = 100
apple_num = 2
orange = 150
orange_num = 3
tax = 1.1
```

（…次頁に続く）

加算・乗算レイヤの実装

（前頁からの続き）

```
# 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)
orange_price = mul_orange_layer.forward(orange, orange_num)
all_price = add_apple_orange_layer.forward(apple_price, orange_price)
price = mul_tax_layer.forward(all_price, tax)
# backward
dprice = 1
dall_price, dtax = mul_tax_layer.backward(dprice)
dapple_price, dorange_price = add_apple_orange_layer.backward(dall_price)
dorange, dorange_num = mul_orange_layer.backward(dorange_price)
dapple, dapple_num = mul_apple_layer.backward(dapple_price)
```

（…次頁に続く）

加算・乗算レイヤの実装

（前頁からの続き）

```
# print
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)
```

加算・乗算レイヤの実装

☆ 実行結果

```
Mounted at /content/drive
price: 715
dApple: 2.2
dApple_num: 110
dOrange: 3.3000000000000003
dOrange_num: 165
dTax: 650
```

◎先週の「余力」の答えは 165 でした

活性化関数レイヤの実装

- ニューラルネット（１） で取り上げた活性化関数のレイヤ（順・逆伝播）を実装する
  - 
  - 

活性化関数レイヤの実装

- ニューラルネット（１） で取り上げた活性化関数のレイヤ（順・逆伝播）を実装する
  - ReLU
  -

活性化関数レイヤの実装

- ニューラルネット（１） で取り上げた活性化関数のレイヤ（順・逆伝播）を実装する
  - ReLU
  - Sigmoid

ReLU レイヤ

☆ ReLU 関数

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

について；

- 微分は

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

(厳密には $x = 0$ では左側微分係数（略）)

- つまり逆伝搬するとき、$x > 0$ のときは上流の値を [　　]、$x \leq 0$ のときは下流への信号はそこで [　　] する

ReLU レイヤ

☆ ReLU 関数

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

について；

- 微分は

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

(厳密には $x = 0$ では左側微分係数（略）)

- つまり逆伝搬するとき、$x > 0$ のときは上流の値を そのまま、$x \leq 0$ のときは下流への信号はそこで [　　] する

ReLU レイヤ

☆ ReLU 関数

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

について；

- 微分は

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

(厳密には $x = 0$ では左側微分係数（略）)

- つまり逆伝搬するとき、$x > 0$ のときは上流の値を **そのまま**、$x \leq 0$ のときは下流への信号はそこで **ストップ** する

ReLU レイヤの実装

☆以下を Colab. の mymodules の中に layers.py という名前で保存する

```
# coding: utf-8
import numpy as np
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
```

ReLU レイヤ

- self.mask という [　　] で、順伝搬の入力 x の要素で 0 以下の場所を True、それ以外を False として保持
- backward では、上で保持している mask を使って、True の場所を 0 に設定する
- ⇒ReLU は [　　] のような機能を持つ

ReLU レイヤ

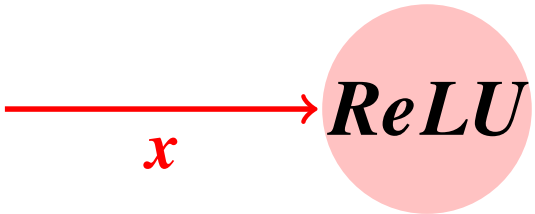
- self.mask という **インスタンス変数** で、順伝搬の入力 x の要素で 0 以下の場所を True、それ以外を False として保持
- backward では、上で保持している mask を使って、True の場所を 0 に設定する
- ⇒ReLU は [　　] のような機能を持つ

ReLU レイヤ

- self.mask という **インスタンス変数** で、順伝搬の入力 x の要素で 0 以下の場所を True、それ以外を False として保持
- backward では、上で保持している mask を使って、True の場所を 0 に設定する
- ⇒ReLU は **スイッチ** のような機能を持つ

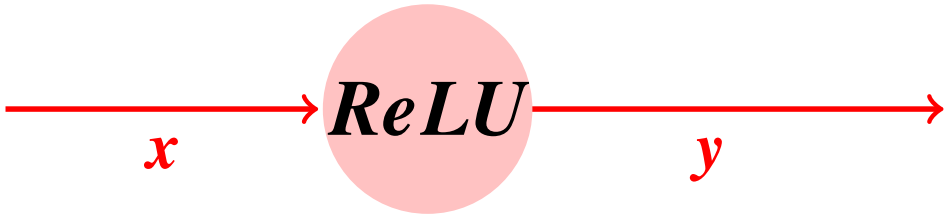
ReLU の逆伝搬

☆ $x > 0$ の場合



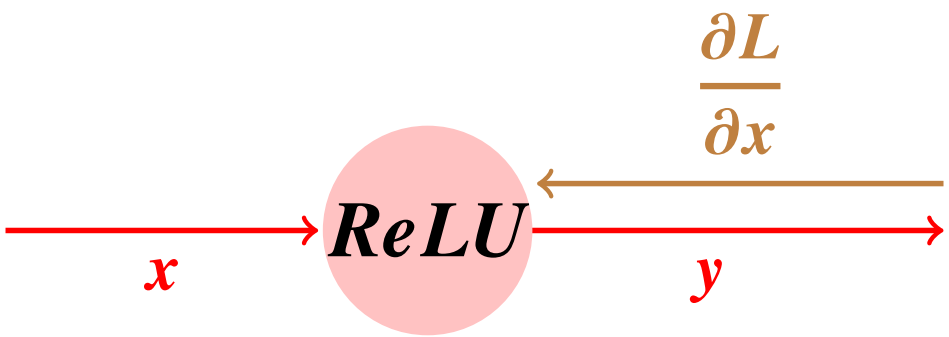
ReLU の逆伝搬

☆ $x > 0$ の場合



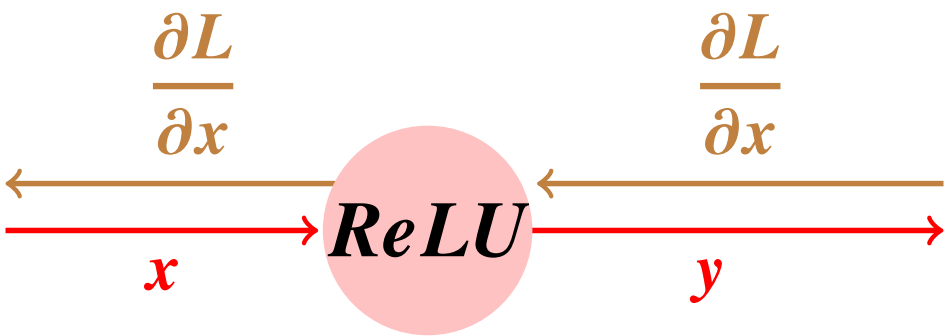
ReLU の逆伝搬

☆ $x > 0$ の場合



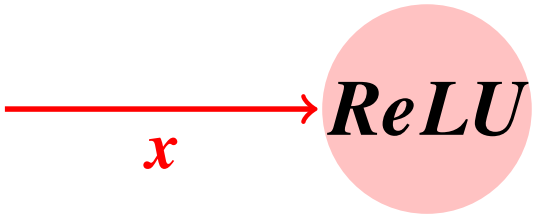
ReLU の逆伝搬

☆ $x > 0$ の場合



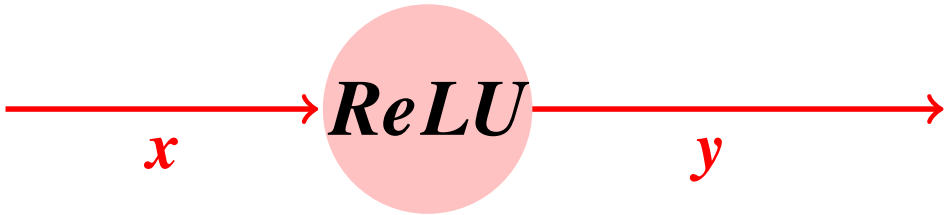
ReLU の逆伝搬

☆ $x \leq 0$ の場合



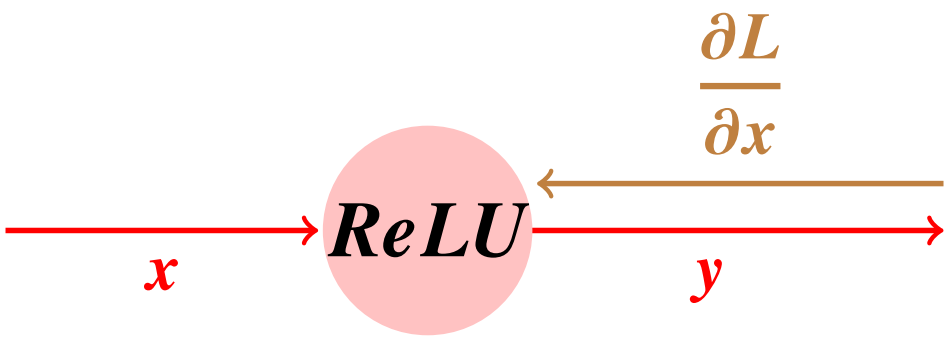
ReLU の逆伝搬

☆ $x \leq 0$ の場合



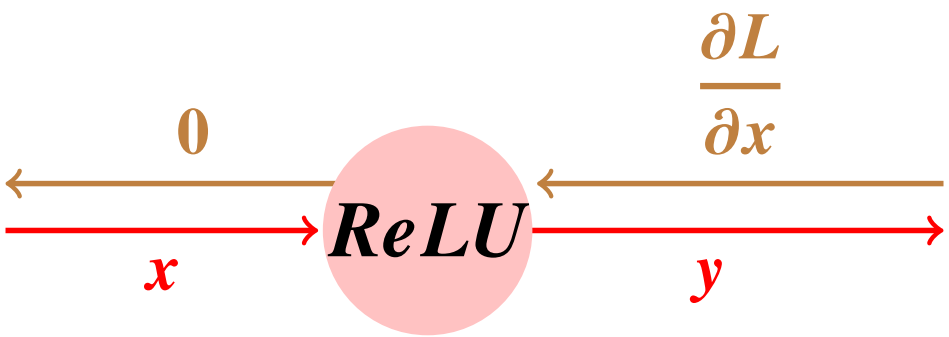
ReLU の逆伝搬

☆ $x \leq 0$ の場合



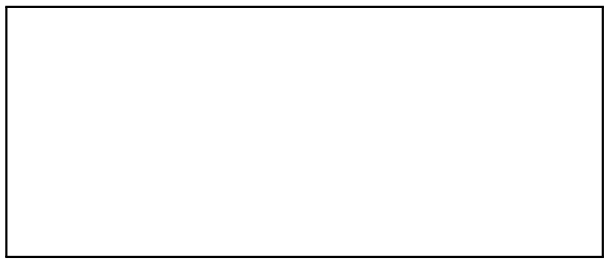
ReLU の逆伝搬

☆ $x \leq 0$ の場合



Sigmoid レイヤ

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

- 微分は  (AI・データサイエンス基礎２の資料参照)
- つまり、逆伝播は順伝播の出力だけで計算できる (!)

Sigmoid レイヤ

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

- 微分は $\frac{dy}{dx} = y(1 - y)$ (AI・データサイエンス基礎２の資料参照)
- つまり、逆伝播は順伝播の出力だけで計算できる (!)

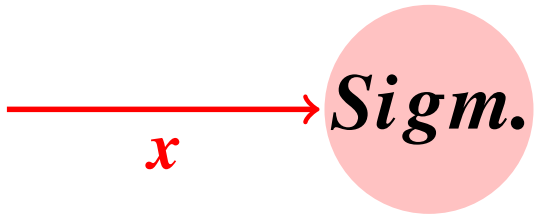
Sigmoid レイヤの実装

☆以下を Colab. の mymodules の中の layers.py に追加する

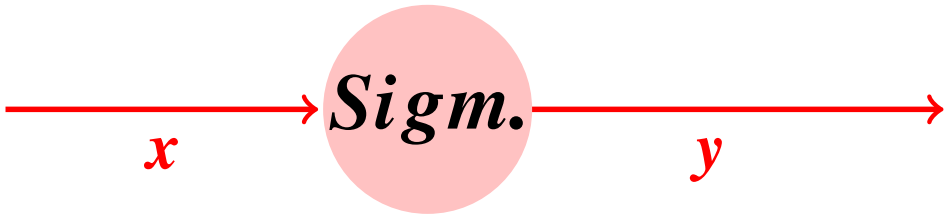
```
class Sigmoid:
    def __init__(self):
        self.out = None
    def forward(self, x):
        out = sigmoid(x)
        self.out = out
        return out
    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```

◎順伝播時に出力をインスタンス変数の self.out に保持しておき、逆伝播時にそれを使っていることに注意 (!)

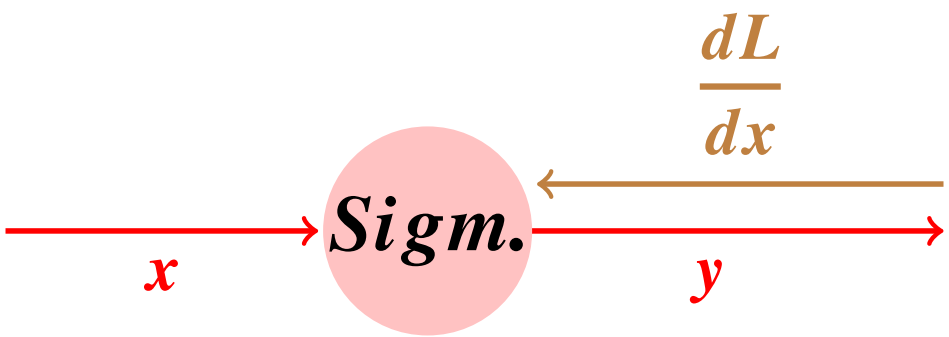
Sigmoid の逆伝搬



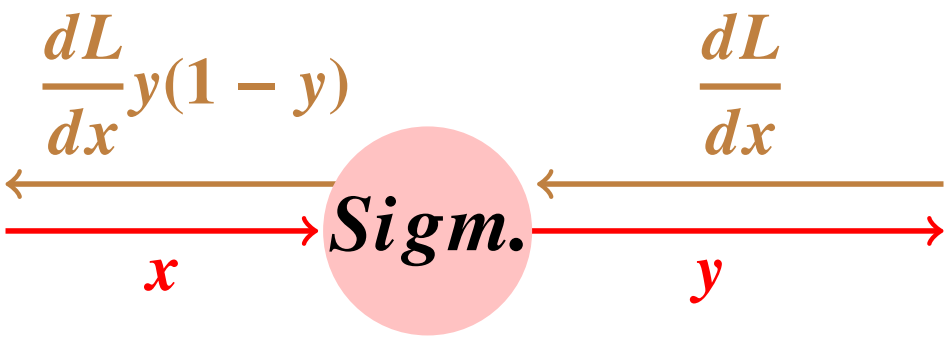
Sigmoid の逆伝搬



Sigmoid の逆伝搬



Sigmoid の逆伝搬



Sigmoid レイヤ（補足）

- $y = \dfrac{1}{1 + exp(-x)}$ を細かく（丁寧に？）
  - x と -1 を掛けて、
  - それの指数関数を求め、($\dfrac{d}{dx} exp(x) = exp(x)$ に注意)
  - それに 1 を加えて
  - 1 をそれで割る ($\dfrac{d}{dx}\dfrac{1}{x} = -x^2$ に注意)

  という計算グラフを作って実装することも可能
  (余力のある人の課題!)

Affine レイヤの実装

- ニューラルネットの順伝搬では
  - 行列の積を計算し
  - 活性化関数で変換し
  - 次の層へ伝搬
- この、行列の積 = [　　] (幾何学用語)
- 次元を揃えるのが重要だった
- NumPy では [　　] で計算していた

Affine レイヤの実装

- ニューラルネットの順伝搬では
  - 行列の積を計算し
  - 活性化関数で変換し
  - 次の層へ伝搬
- この、行列の積 = Affine 変換 (幾何学用語)
- 次元を揃えるのが重要だった
- NumPy では [ ] で計算していた

Affine レイヤの実装

- ニューラルネットの順伝搬では
  - 行列の積を計算し
  - 活性化関数で変換し
  - 次の層へ伝搬
- この、行列の積 = Affine 変換 (幾何学用語)
- 次元を揃えるのが重要だった
- NumPy では np.dot で計算していた

Affine 変換の計算グラフ

☆ 次頁の計算グラフで

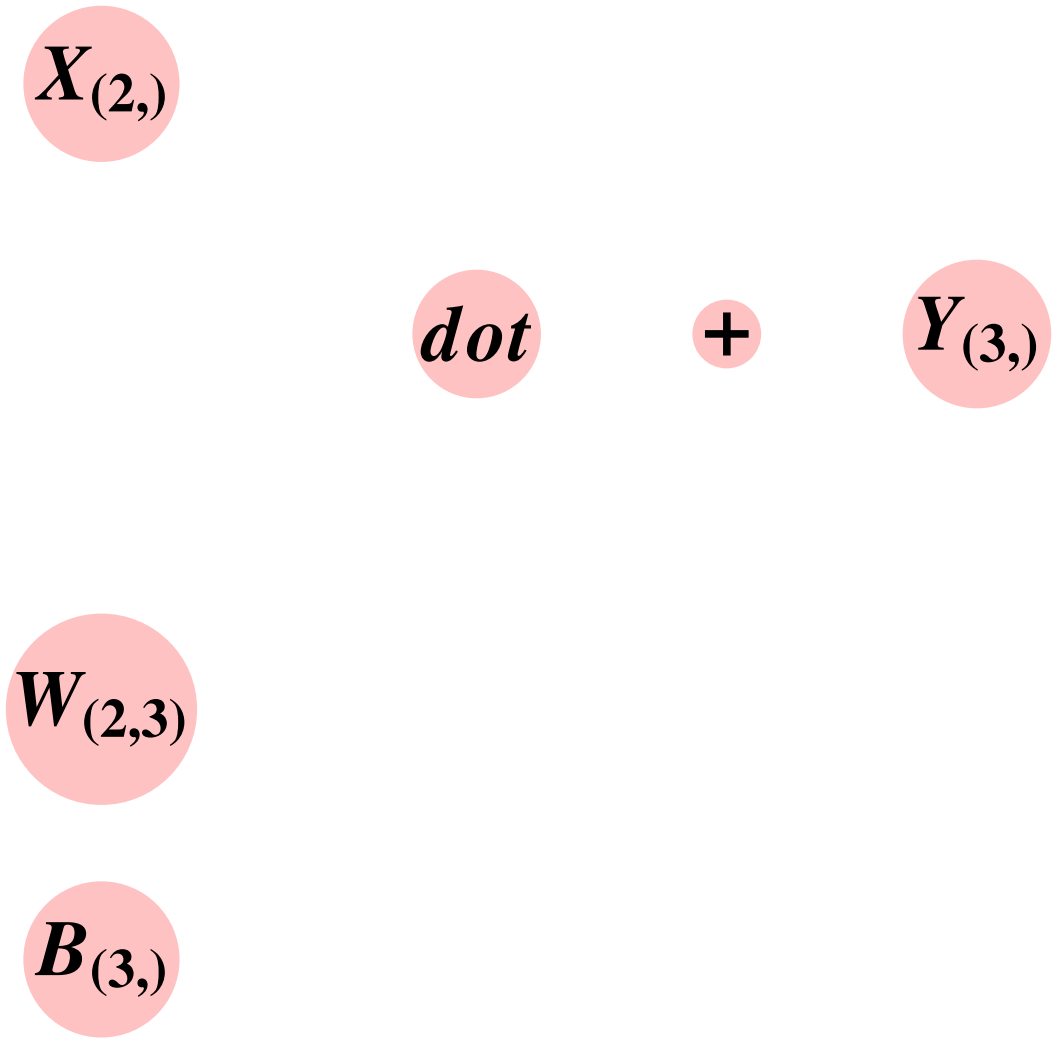
- X, W, B, Y は
  - X は入力（ここでは 2x1 次元）
  - W は重み（ここでは 2x3 次元）
  - B はバイアス（ここでは 3x1 次元）
  - Y は出力（ここでは 3x1 次元）

Affine 変換の計算グラフ

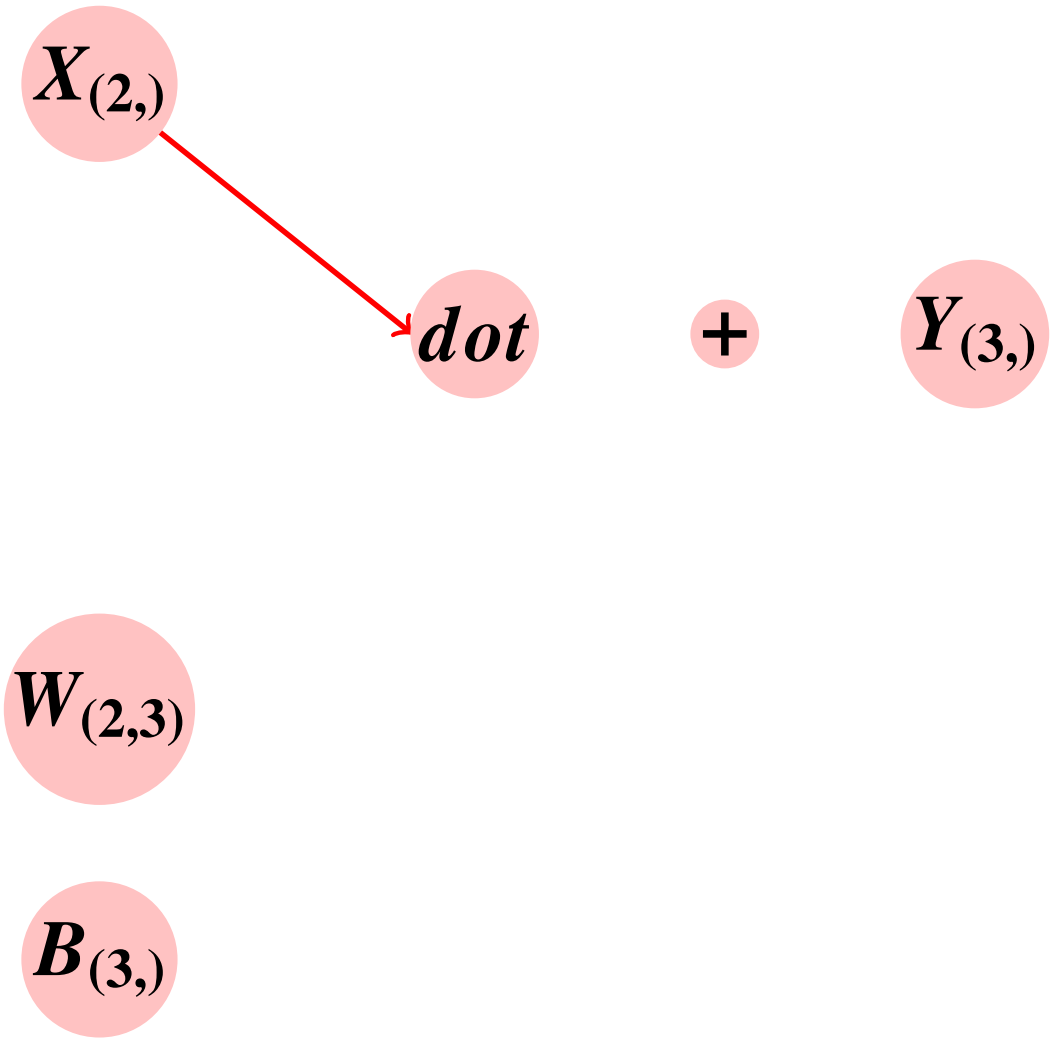
☆ 次頁の計算グラフで

- X, W, B, Y は **行列（多次元配列）**
  - X は入力（ここでは 2x1 次元）
  - W は重み（ここでは 2x3 次元）
  - B はバイアス（ここでは 3x1 次元）
  - Y は出力（ここでは 3x1 次元）

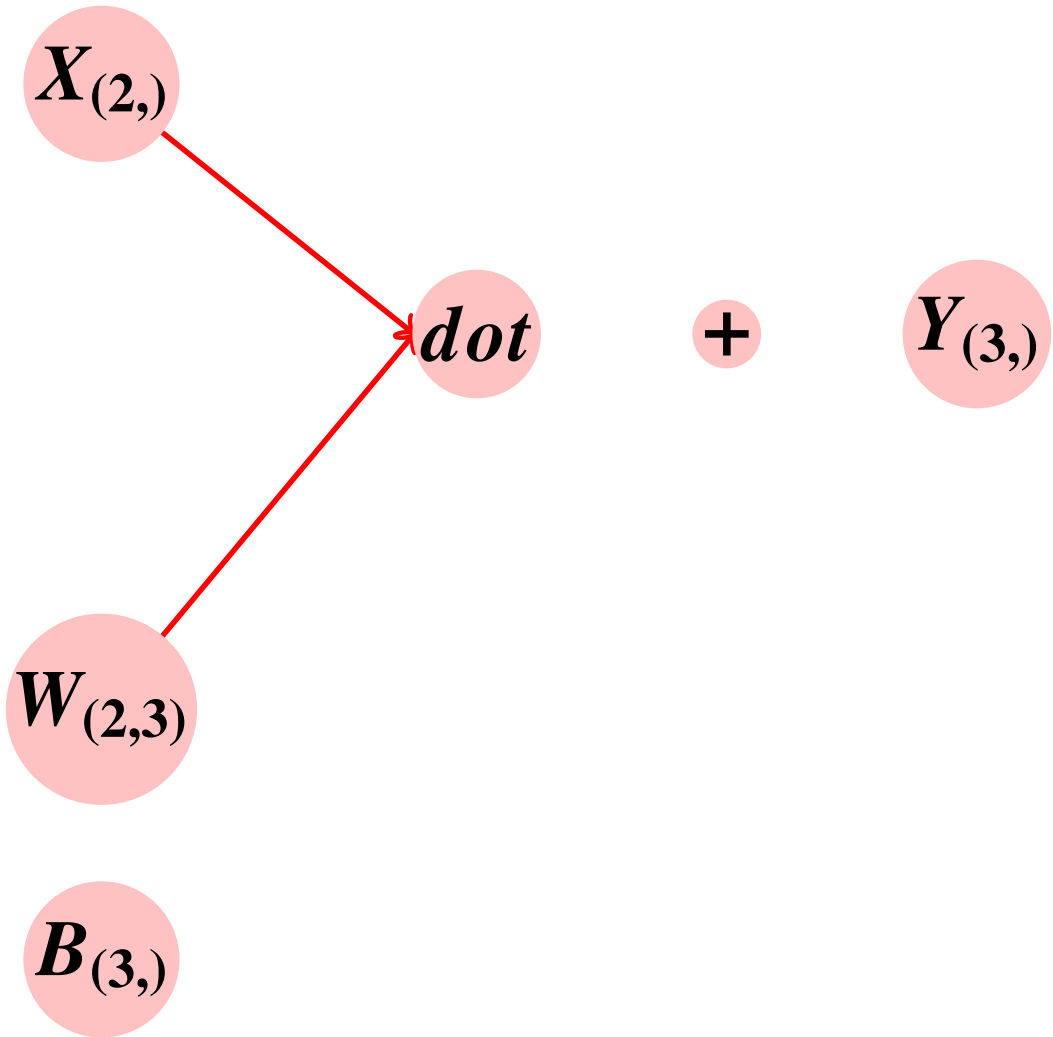
Affine 変換の計算グラフ



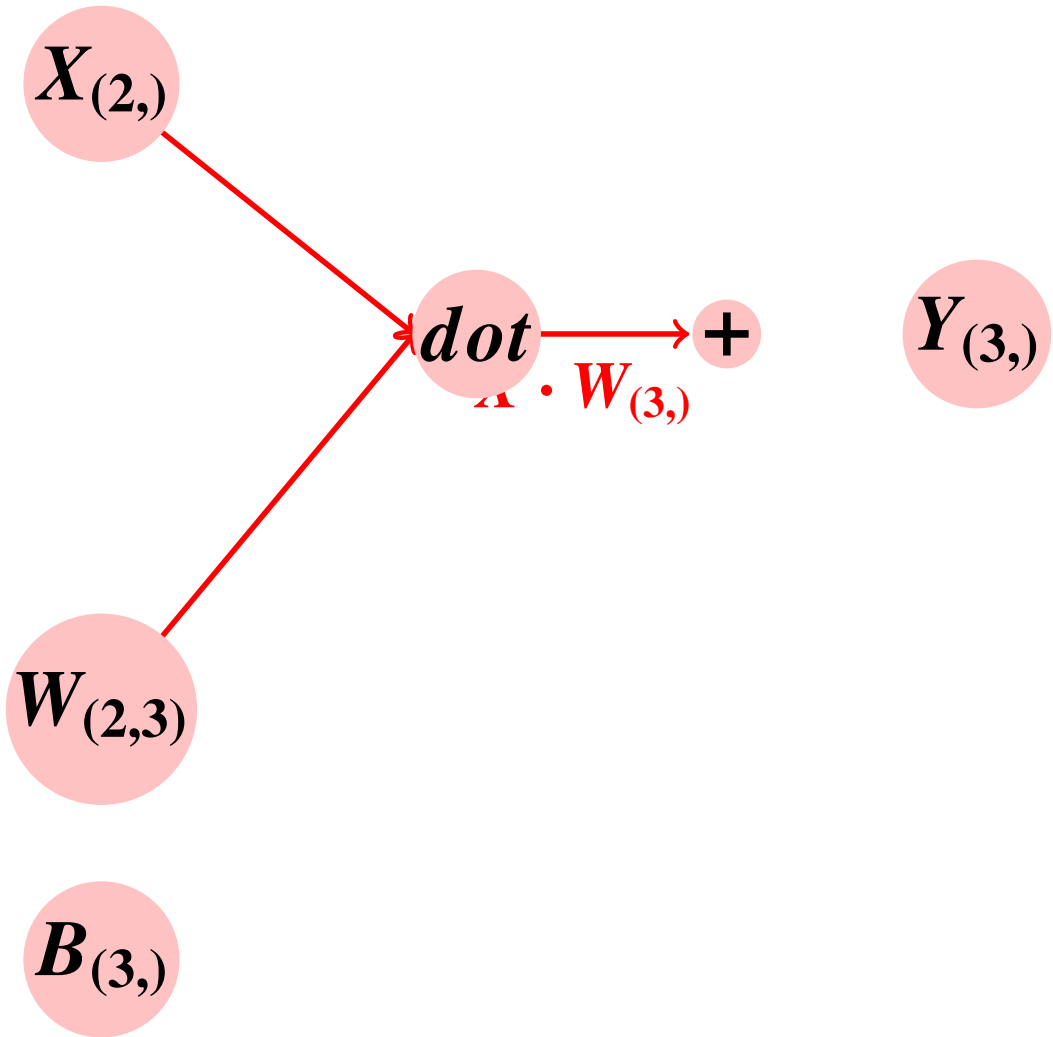
Affine 変換の計算グラフ



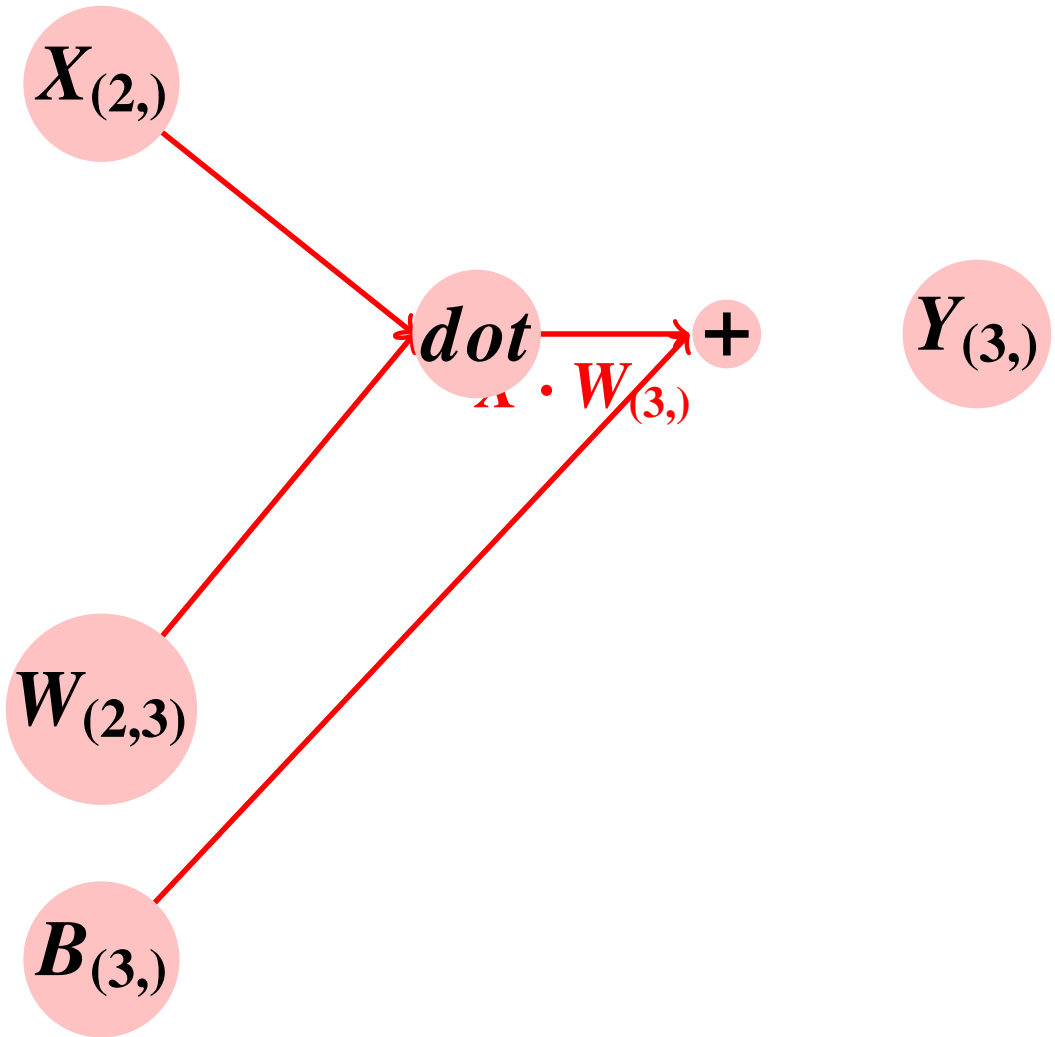
Affine 変換の計算グラフ



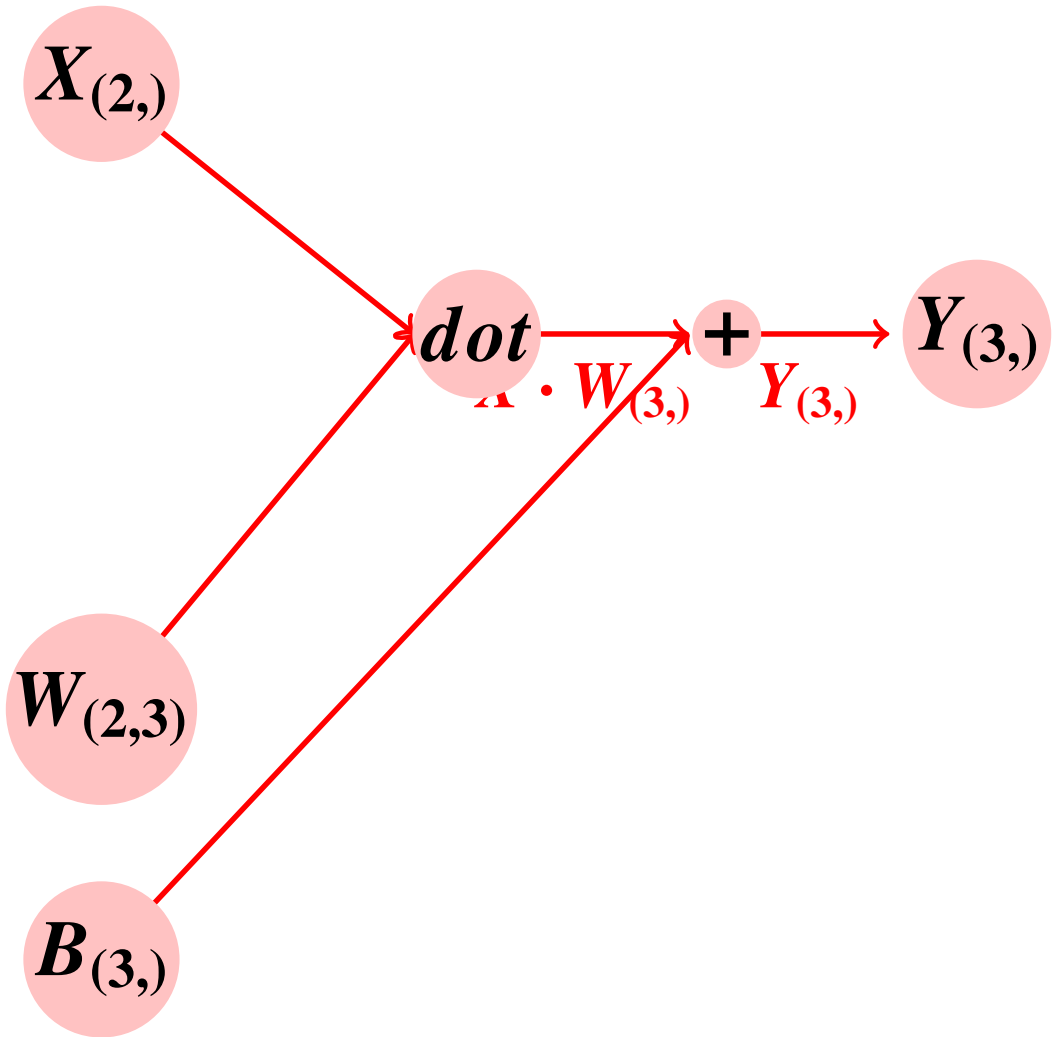
Affine 変換の計算グラフ



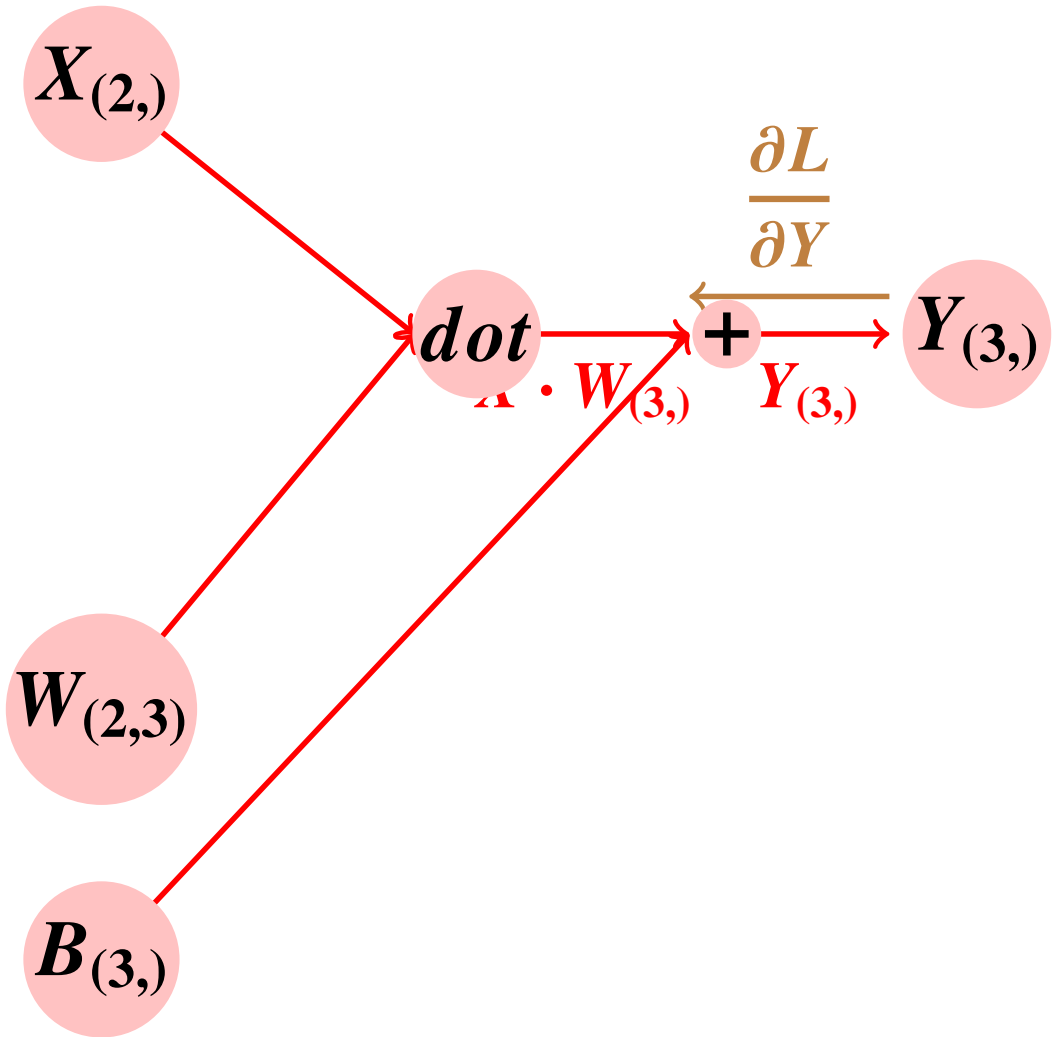
Affine 変換の計算グラフ



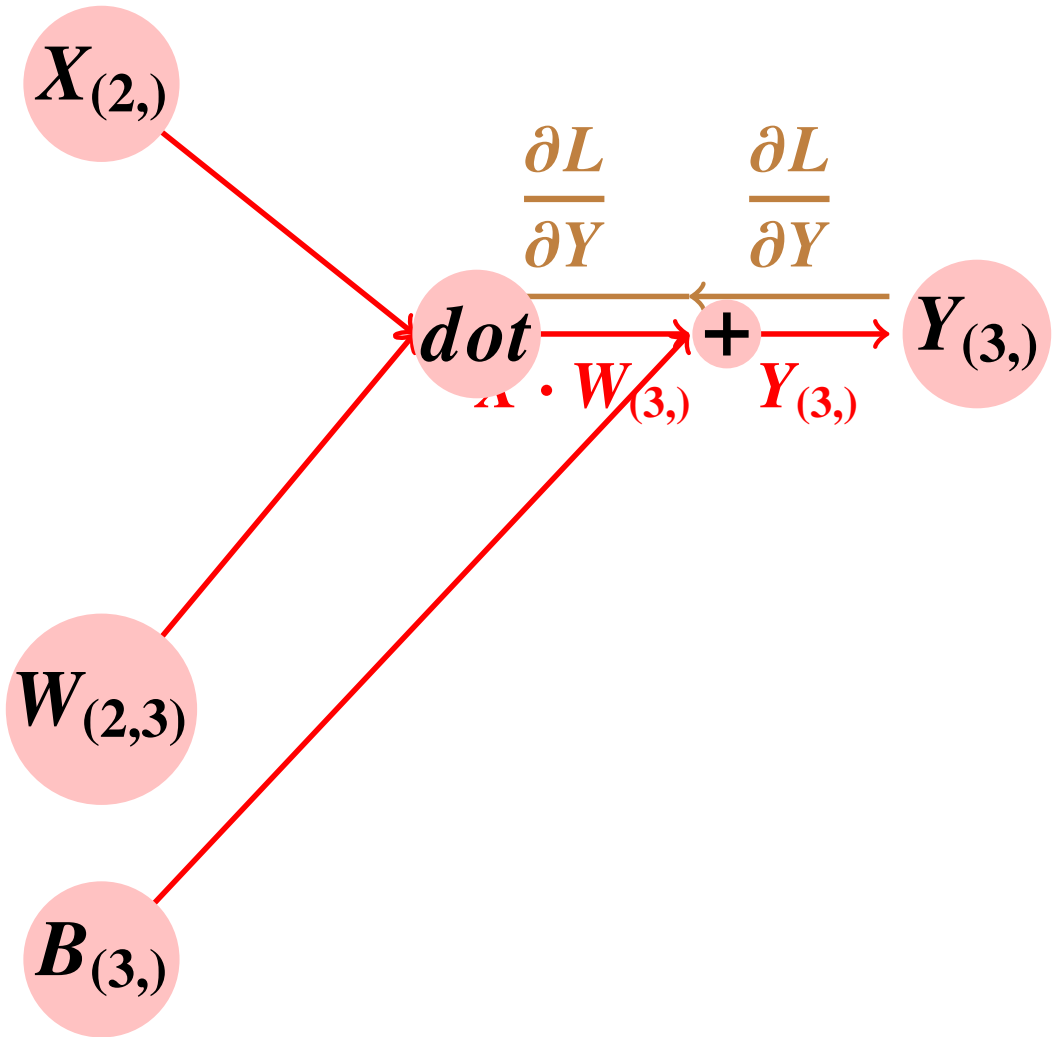
Affine 変換の計算グラフ



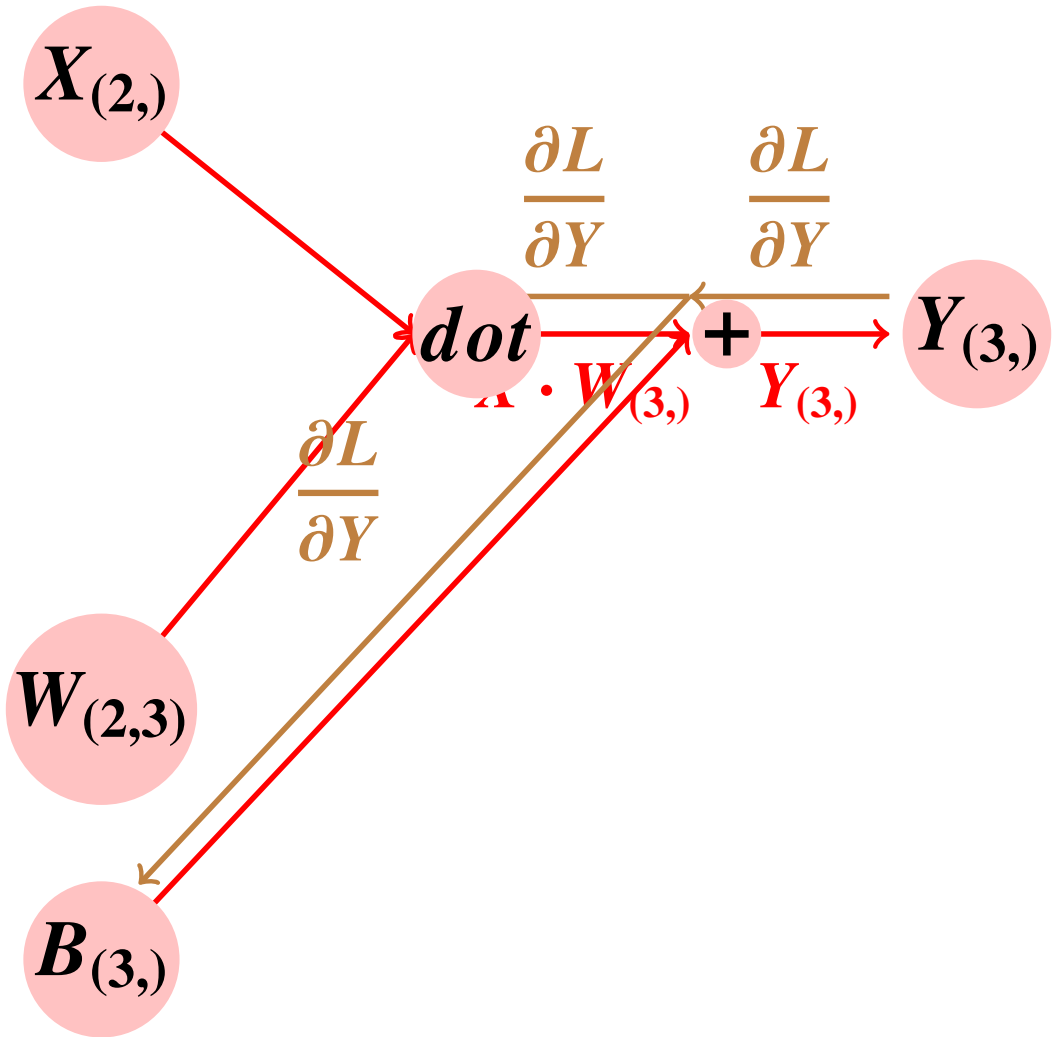
Affine 変換の計算グラフ



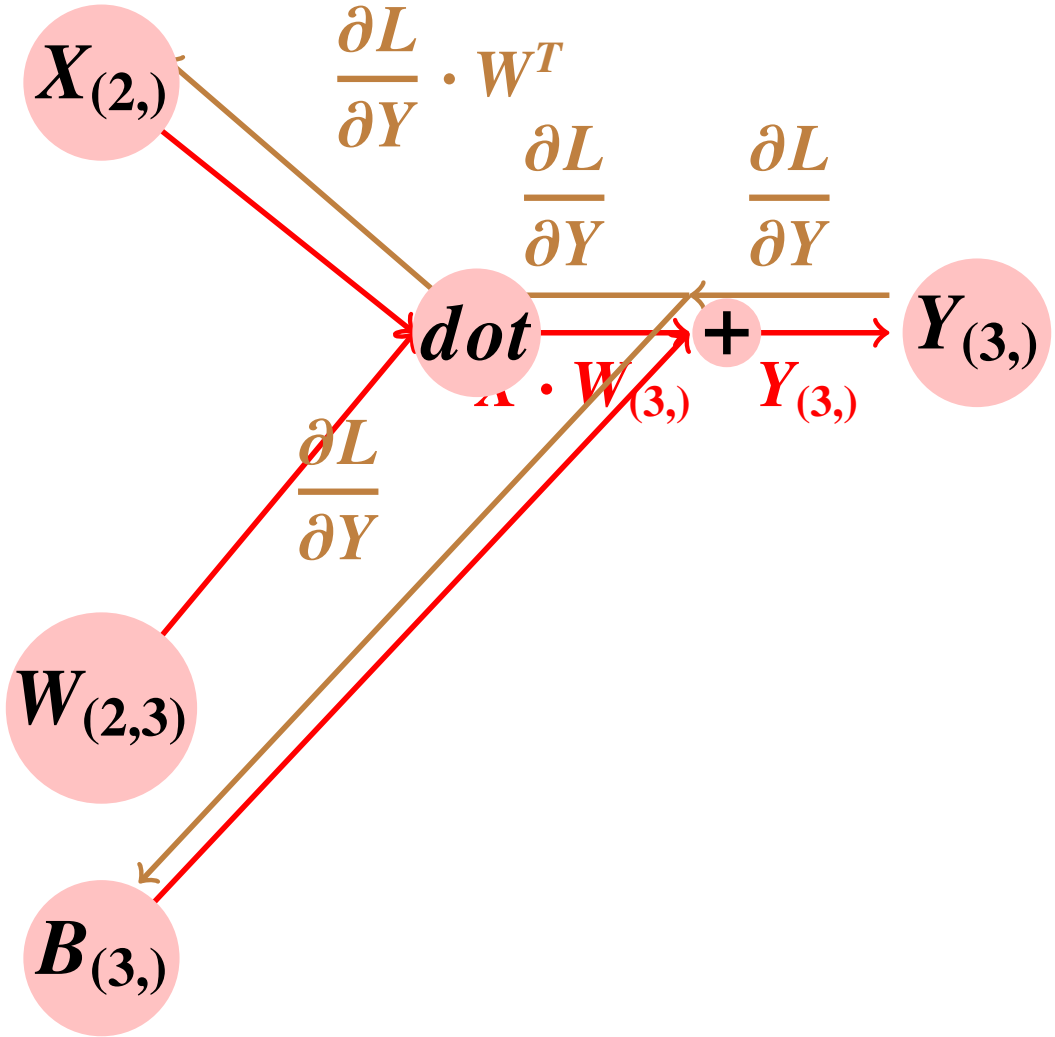
Affine 変換の計算グラフ



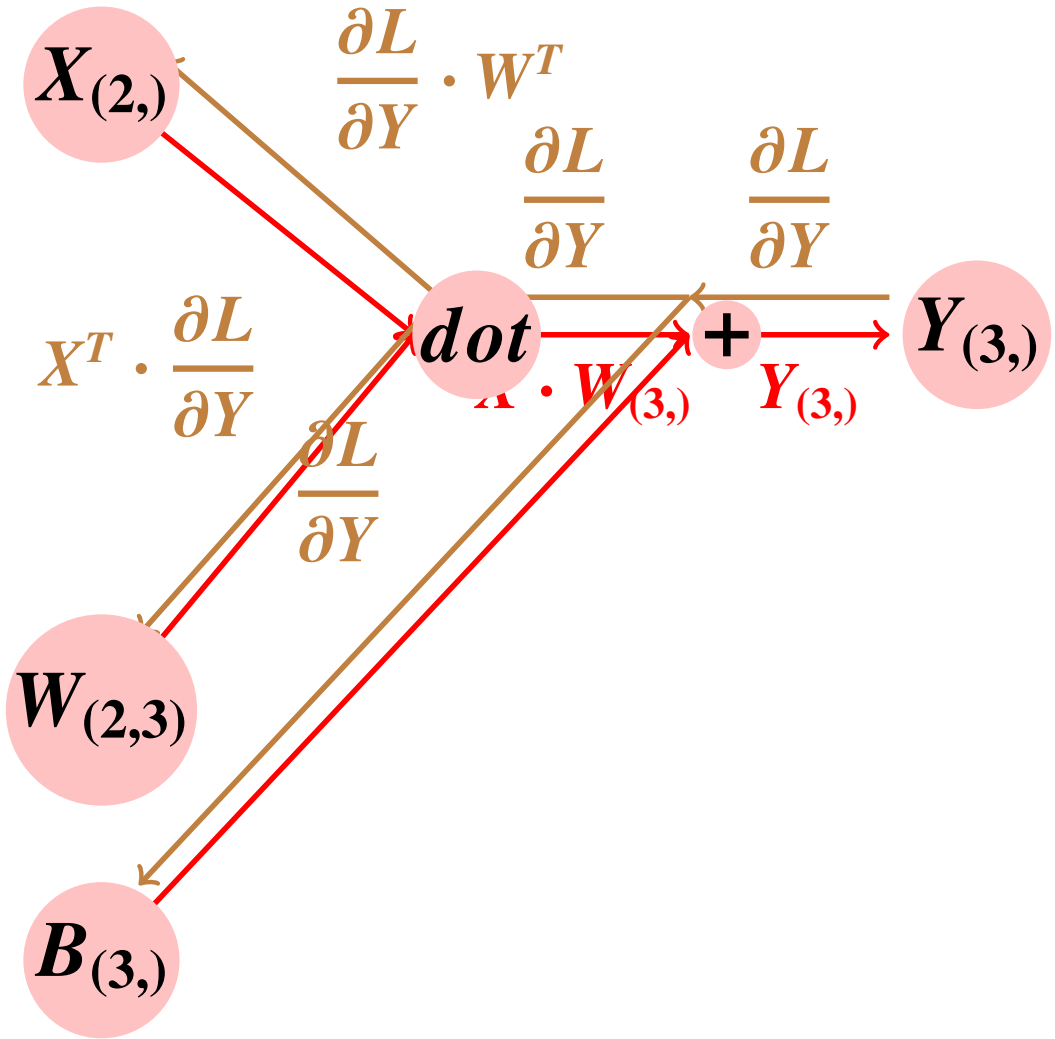
Affine 変換の計算グラフ



Affine 変換の計算グラフ



Affine 変換の計算グラフ



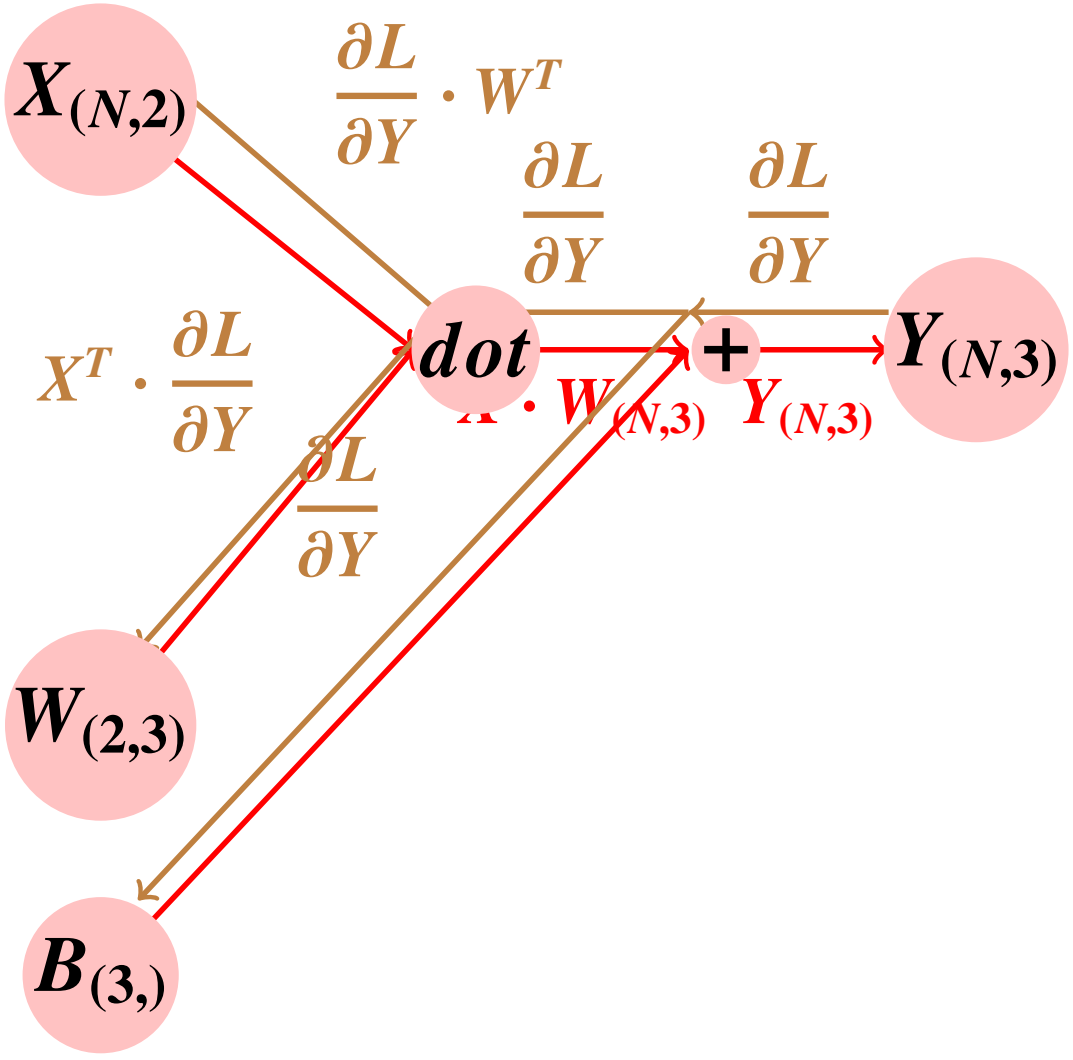
バッチ版 Affine レイヤの実装

- 前掲のレイヤは入力 X は１つのデータを仮定
- Ｎ個のデータを纏めたバッチ版にして高速化をはかる
- 出力が  次元になることに注意 (!)

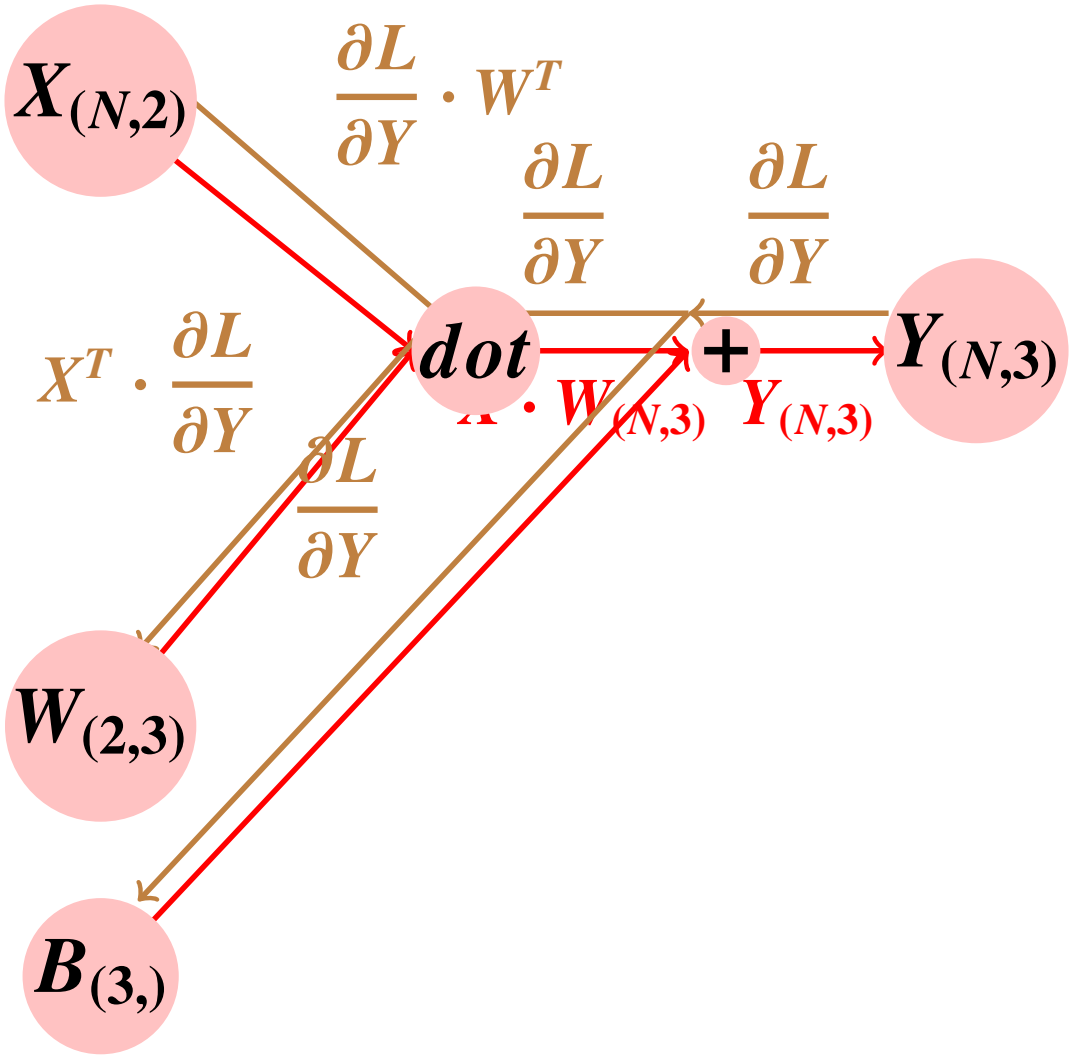
バッチ版 Affine レイヤの実装

- 前掲のレイヤは入力 X は１つのデータを仮定
- N 個のデータを纏めたバッチ版にして高速化をはかる
- 出力が Nx3 次元になることに注意 (!)

バッチ版 Affine 変換の計算グラフ



バッチ版 Affine 変換の計算グラフ



バッチ版 Affine 変換の実装

☆以下を mymodule の中の layers.py に追加する

```
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
```

（…次頁に続く）

バッチ版 Affine 変換の実装

（前頁からの続き）

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

Softmax-with-Loss レイヤ

- Softmax: 出力層として、正規化した（総和が１になる）値のベクトルを出力する（ニューラルネット（２） 参照）
  - MNIST の手書き分類だと 10 要素
- 実は [　　] のときには正規化する必要はなく、その前段階（スコア）が一番高いものだけに注目すれば良い
- [　　] のときにはこのレイヤが必要、ということ
- 損失関数である交差エントロピー誤差と合せて１つのレイヤーとして実装

Softmax-with-Loss レイヤ

- Softmax: 出力層として、正規化した（総和が１になる）値のベクトルを出力する（ニューラルネット（２） 参照）
  - MNIST の手書き分類だと 10 要素
- 実は 推論 のときには正規化する必要はなく、その前段階（スコア）が一番高いものだけに注目すれば良い
- [　　] のときにはこのレイヤが必要、ということ
- 損失関数である交差エントロピー誤差と合せて１つのレイヤーとして実装

Softmax-with-Loss レイヤ

- Softmax: 出力層として、正規化した（総和が１になる）値のベクトルを出力する（ニューラルネット（２） 参照）
  - MNIST の手書き分類だと 10 要素
- 実は 推論 のときには正規化する必要はなく、その前段階（スコア）が一番高いものだけに注目すれば良い
- 学習 のときにはこのレイヤが必要、ということ
- 損失関数である交差エントロピー誤差と合せて１つのレイヤーとして実装

Softmax-with-Loss レイヤ

☆次頁の計算グラフ（３入力１出力）において

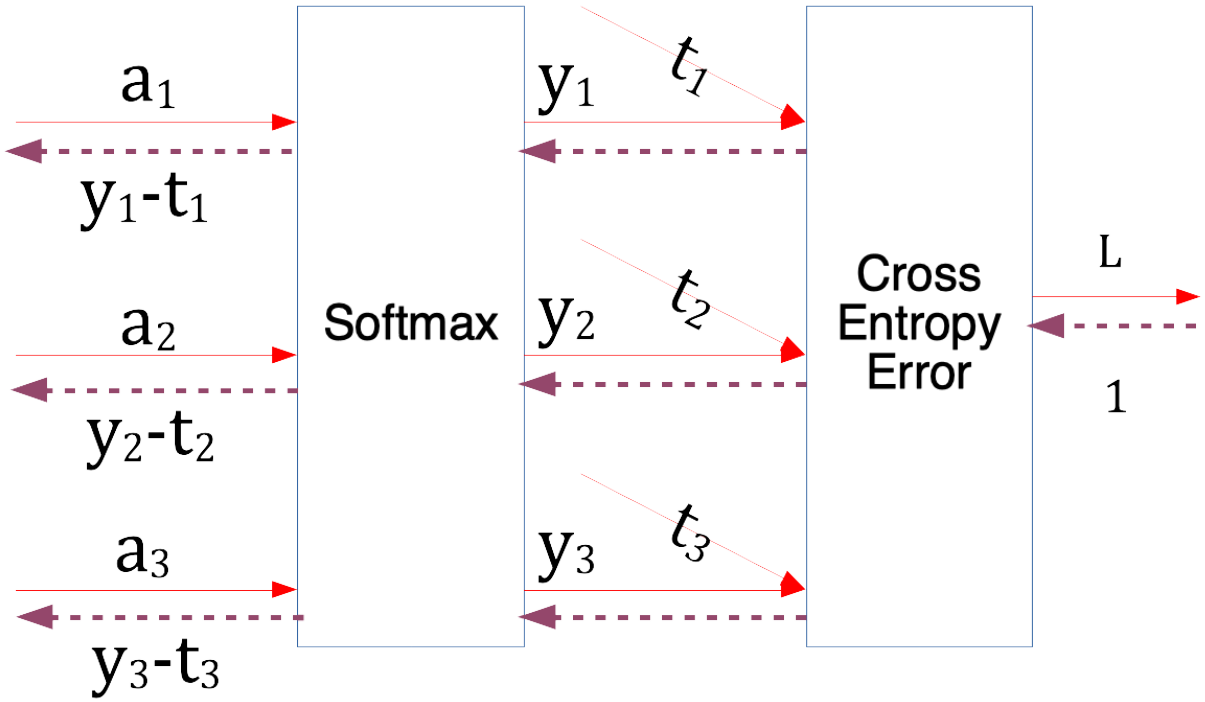
- 入力 (a_1, a_2, a_3) を正規化して (y_1, y_2, y_3) を出力
- その出力と教師ラベルの (t_1, t_2, t_3) を受け取り損失 L を出力する
- 逆伝搬は [　　] となる（重要！）

Softmax-with-Loss レイヤ

☆次頁の計算グラフ（３入力１出力）において

- 入力 (a_1, a_2, a_3) を正規化して (y_1, y_2, y_3) を出力
- その出力と教師ラベルの (t_1, t_2, t_3) を受け取り損失 L を出力する
- 逆伝搬は $\boxed{(y_1 - t_1, y_2 - t_2, y_3 - t_3)}$ となる（重要！）

Softmax-with-Loss レイヤ



Softmax-with-Loss レイヤの実装

☆以下を mymodule の中の layers.py に追加する

```
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
```

（…次頁に続く）

Softmax-with-Loss レイヤの実装

（前頁からの続き）

```
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
```

ニューラルネット学習の全体図

☆前提：適応可能な重みとバイアスがあり、調整できる

1. [　　]: 訓練データの中からランダムに一部のデータを選ぶ
2. [　　]: 損失関数の勾配を求める
3. [　　]: 重みパラメータを勾配方向に微小量更新する
4. [　　]: 上を繰り返す

ニューラルネット学習の全体図

☆前提：適応可能な重みとバイアスがあり、調整できる

1. ミニバッチ: 訓練データの中からランダムに一部のデータを選ぶ
2. : 損失関数の勾配を求める
3. : 重みパラメータを勾配方向に微小量更新する
4. : 上を繰り返す

ニューラルネット学習の全体図

☆前提：適応可能な重みとバイアスがあり、調整できる

1. ミニバッチ: 訓練データの中からランダムに一部のデータを選ぶ
2. 勾配の算出: 損失関数の勾配を求める
3. : 重みパラメータを勾配方向に微小量更新する
4. : 上を繰り返す

ニューラルネット学習の全体図

☆前提：適応可能な重みとバイアスがあり、調整できる

1. **ミニバッチ**: 訓練データの中からランダムに一部のデータを選ぶ
2. **勾配の算出**: 損失関数の勾配を求める
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更新する
4. ＿＿＿＿＿＿: 上を繰り返す

ニューラルネット学習の全体図

☆前提：適応可能な重みとバイアスがあり、調整できる

1. **ミニバッチ**: 訓練データの中からランダムに一部のデータを選ぶ
2. **勾配の算出**: 損失関数の勾配を求める
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更新する
4. **繰り返し**: 上を繰り返す

バックプロパゲーション対応ニューラルネット

☆以下、tl_nn.py のリンク先を Colab. の中の mymodules に tl_nn.py という名前で保存する

```
# coding: utf-8
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
import numpy as np
sys.path.append(bd)
import numpy as np
from layers import *
from gradient import numerical_gradient
from collections import OrderedDict
```

（…次頁に続く）

バックプロパゲーション対応ニューラルネット

（前頁からの続き）

```
class TwoLayerNet:
    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() # ReLU を使う
        self.layers['Affine2'] = \
            Affine(self.params['W2'], self.params['b2'])
        self.lastLayer = SoftmaxWithLoss()
```

（次頁に続く）

バックプロパゲーション対応ニューラルネット

（前頁からの続き）

```
    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)
        return x
    # x:入力データ, t:教師データ
    def loss(self, x, t):
        y = self.predict(x)
        return self.lastLayer.forward(y, t)
    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
```

（…次頁に続く）

バックプロパゲーション対応ニューラルネット

（前頁からの続き）

```
# 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
```

（…次頁に続く）

バックプロパゲーション対応ニューラルネット

（前頁からの続き）

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

バックプロパゲーションを使った学習

- ☆準備：念のため、これまで作ってきた関数をまとめてあるので、functions.py のリンク先を Colab の mymodules に（上書き）保存する
- layers.py も用意してあるので、 layers.py の編集がうまくいってない人は、リンク先を Colab の mymodules に（上書き）保存する

バックプロパゲーションを使った学習

☆以下、 tr_nn5.py のリンク先を Colab. にコピー＆ペーストして実行する

```
# coding: utf-8
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
sys.path.append(bd)
import numpy as np
from mnist import load_mnist
from tl_nn 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)
```

（…次頁に続く）

バックプロパゲーションを使った学習

（前頁からの続き）

```
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)
```

（…次頁に続く）

バックプロパゲーションを使った学習

（前頁からの続き）

```
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)
```

（…次頁に続く）

バックプロパゲーションを使った学習

（前頁からの続き）

```
# 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)
```

バックプロパゲーションを使った学習

実行結果

```
Mounted at /content/drive
Mounted at /content/drive
0.11633333333333333 0.1222
0.9045 0.9083
0.9243666666666667 0.9273
0.9397166666666666 0.9403
0.9470833333333334 0.9487
0.9534166666666667 0.9509
0.9593333333333334 0.9571
0.9623333333333334 0.9601
0.9644833333333334 0.9607
0.9692833333333334 0.9655
0.9707 0.9665
0.9729666666666666 0.9683
0.9741666666666666 0.9688
0.9763833333333334 0.9701
0.9758833333333333 0.9692
0.9773 0.9688
0.98035 0.97
```

おしまい

質問・コメント等あれば、何でもお気軽に
aipr@e-chan.jp に
メールください！