Deep learning

May.22/2020

Ochanomizu University
Kimu Tsuri

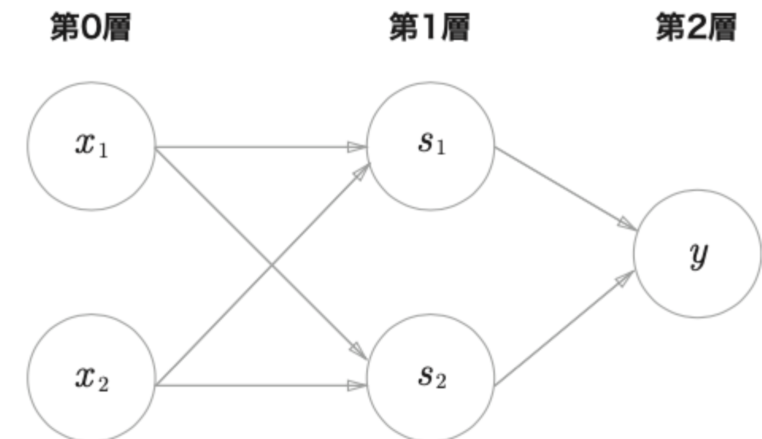
3章 ニューラルネットワーク

■ 前回まで

- パーセプトロンについて学んだ
  - 利点

    層を重ね合わせることで複雑な表現が可能
  - 欠点

    人間が適切な重みを決める必要がある

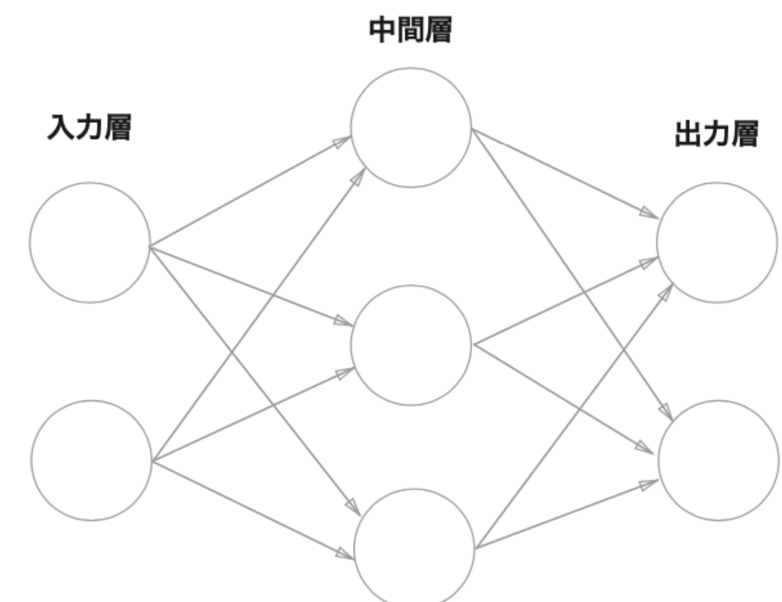


■ 今回

- ニューラルネットワークの導入
  - 利点

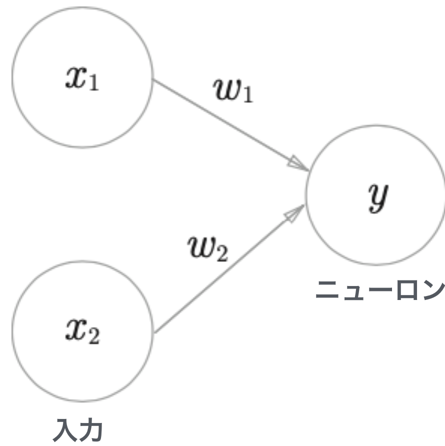
    適切な重みをパラメータから学習できる

2層ニューラルネットワークの例



ニューラルネットワークのイントロ

■ パーセプトロンの復習



出力

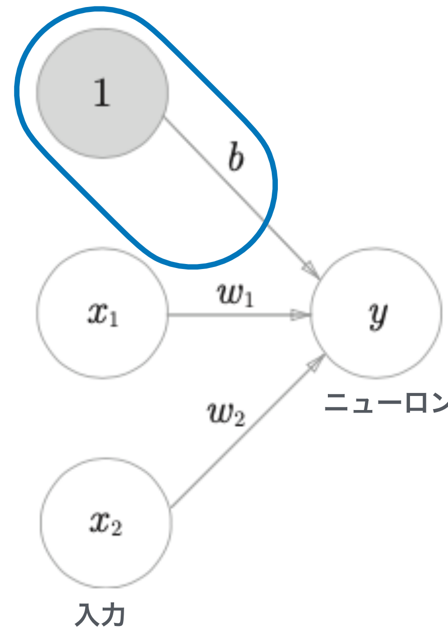
$$y = \begin{cases} 0 & (b + w_1 x_1 + w_2 x_2 \leqq 0) \\ 1 & (b + w_1 x_1 + w_2 x_2 > 0) \end{cases}$$

重み x 入力

バイアス

b x 1 = 重み x 入力

■ バイアスを明示



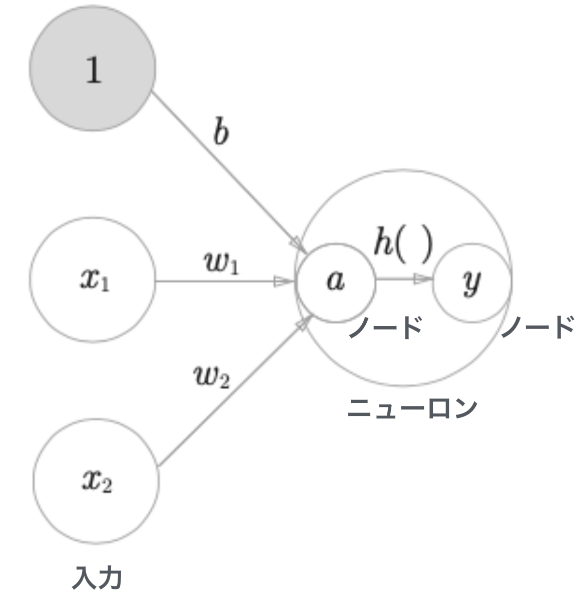
出力

$$y = h(b + w_1 x_1 + w_2 x_2)$$

活性化関数

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

■ 活性化関数を明示



出力

$$y = h(a)$$

$$a = b + w_1 x_1 + w_2 x_2$$

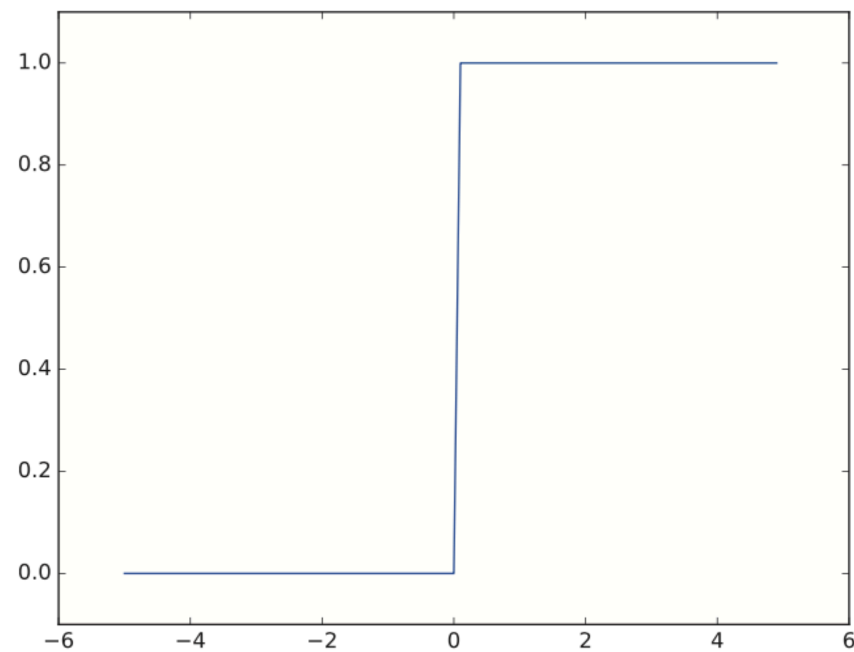
活性化関数

■ ステップ関数

0 or 1 のみ

活性化関数

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



Pythonへ実装例（step_function.py）

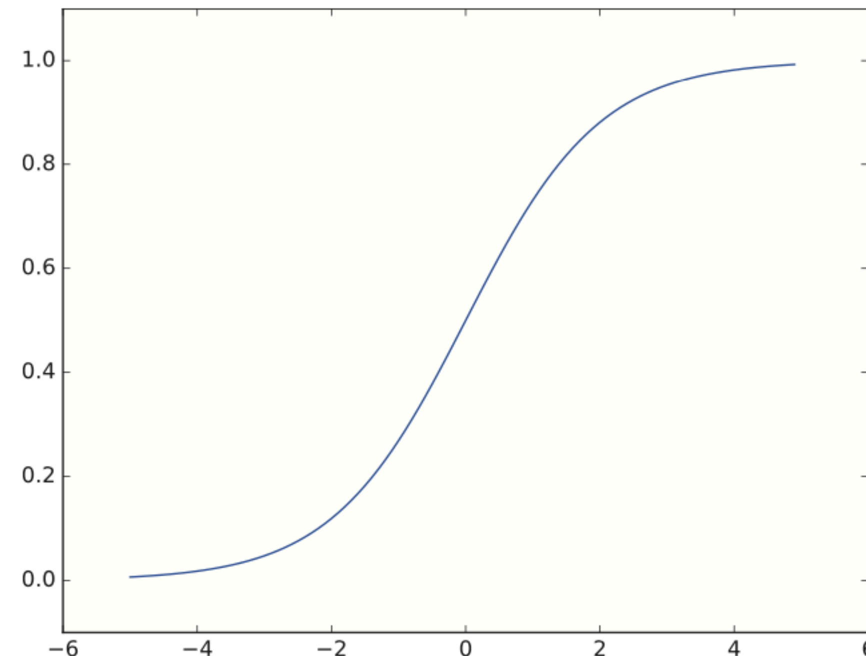
```
def step_function(x):
    return np.array(x > 0, dtype=np.int)

X = np.arange(-5.0, 5.0, 0.1)
Y = step_function(X)
```

■ シグモイド関数

滑らか！
0.78123 のような
値も取れる

活性化関数

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



Pythonへ実装例（sigmoid.py）

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

X = np.arange(-5.0, 5.0, 0.1)
Y = sigmoid(X)
```

活性化関数

■ ステップ関数

0 or 1 のみ

活性化関数

h

1.0
0.8
0.6
0.4
0.2
0.0

Py

```
de
X = np.arange(-5.0, 5.0, 0.1)
Y = step_function(X)
```

■ シグモイド関数

滑らか！
0.78123 のような
値も取れる

活性化関数

```
X = np.arange(-5.0, 5.0, 0.1)
Y = sigmoid(X)
```

ステップ関数とシグモイド関数の共通点

→ 非線形関数

ニューラルネットワークの活性化関数で、線形関数を使うと？

活性化関数　$h(x) = cx$

出力　$y = h(h(h(x))) = c \times c \times c \times x = ax \quad (c^3 = a)$

層を重ねる恩恵を得るためには、

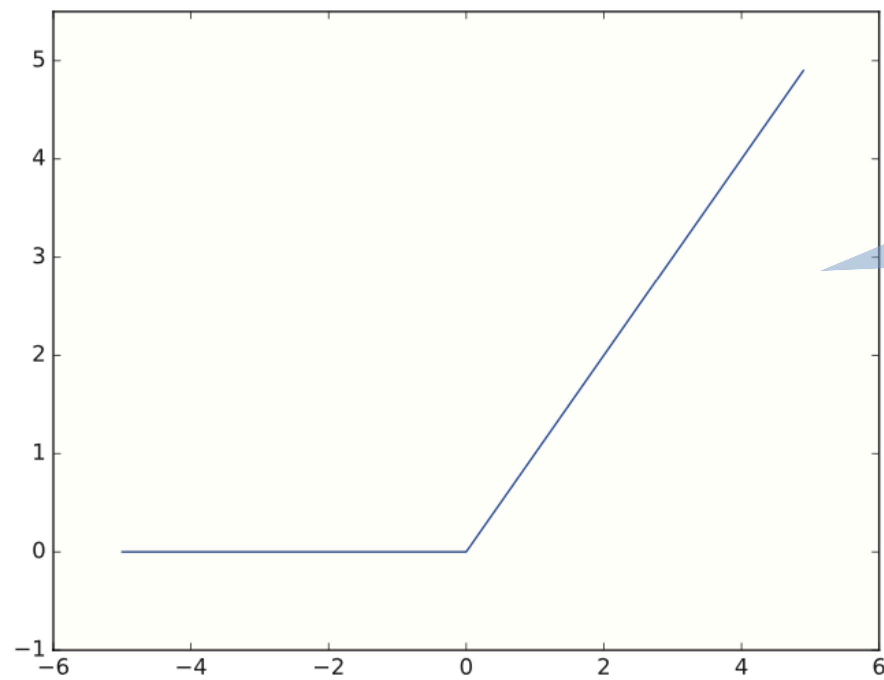
活性化関数に非線形関数を使う必要がある

活性化関数

■ ReLU関数（Rectified Linear Unit）

活性化関数

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



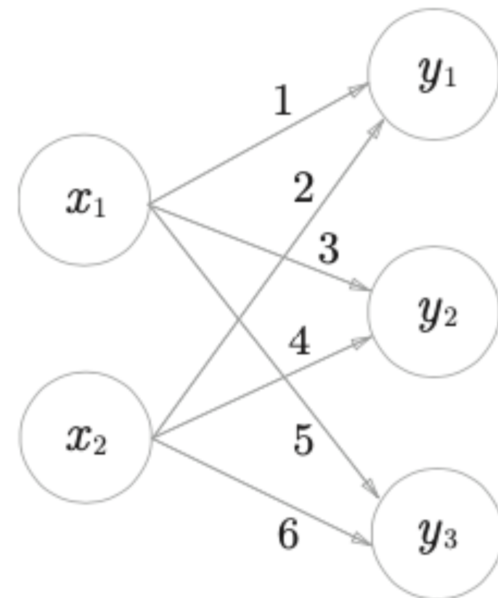
ここは線形

Pythonへ実装例（relu.py）

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

x = np.arange(-5.0, 5.0, 0.1)
y = relu(x)
```

ニューラルネットワークの実装

- ニューラルネットワークの行列の積
  - NumPy行列を使って簡単に表すことができる



$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

入力	重み		ニューロン
$\mathbf{X}$	$\mathbf{W}$	$=$	$\mathbf{Y}$
2	2 × 3		3

一致

Pythonへ実装例（neuralnet_matrix.py）

```
import numpy as np

X = np.array([1, 2])
W = np.array([[1, 3, 5], [2, 4, 6]])

Y = np.dot(X, W)
print(Y)
```

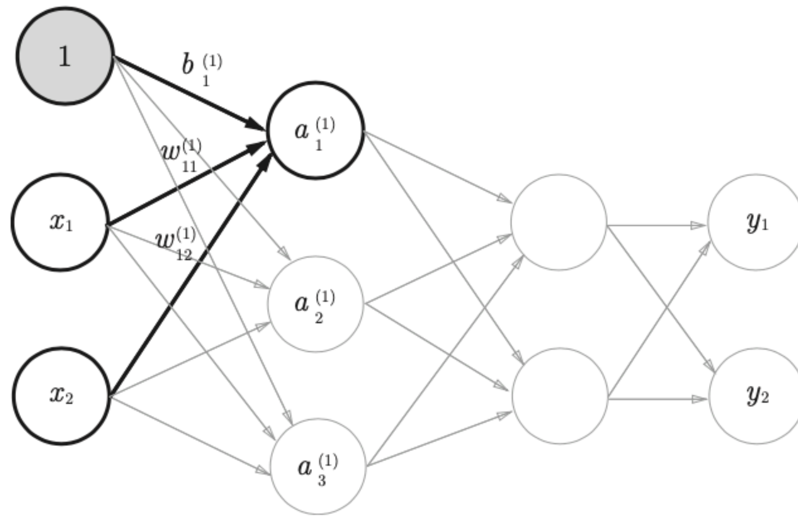
実行 →

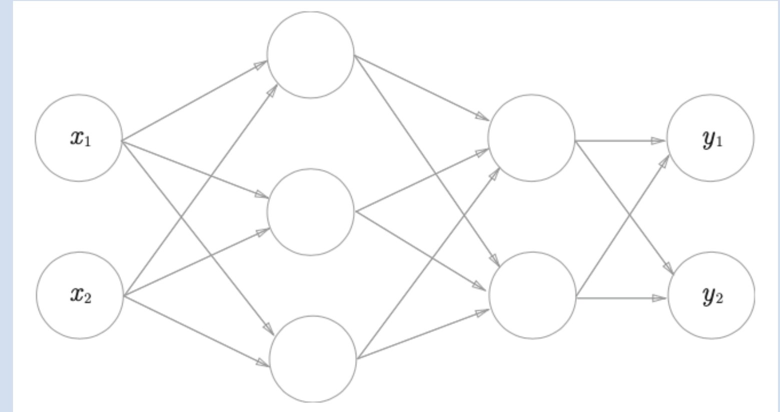
```
[ 5 11 17]
```

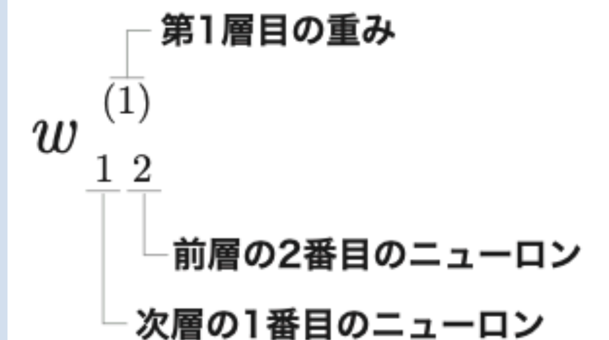
ニューラルネットワークの実装

■ 3層ニューラルネットワーク

① 第１層目の１番目のニューロンへの伝達を入れる







まず、成分で表記してみると

$$a_1^{(1)} = w_{11}^{(1)} x_1 + w_{12}^{(1)} x_2 + b_1^{(1)}$$

重み x 入力　　バイアス

これを行列でまとめる

$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

$$\mathbf{A}^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} x_1 & x_2 \end{pmatrix}$$

$$\mathbf{B}^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

$$\mathbf{W}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

行列表記できたので、Pythonに実装してみる

Pythonの実装例（neuralnet_1layer.py）

```
import numpy as np

X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

A1 = np.dot(X, W1) + B1
print(A1)
```

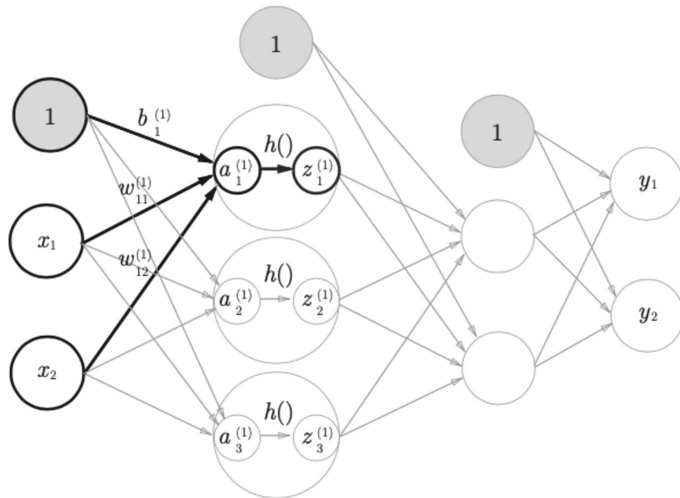
実行

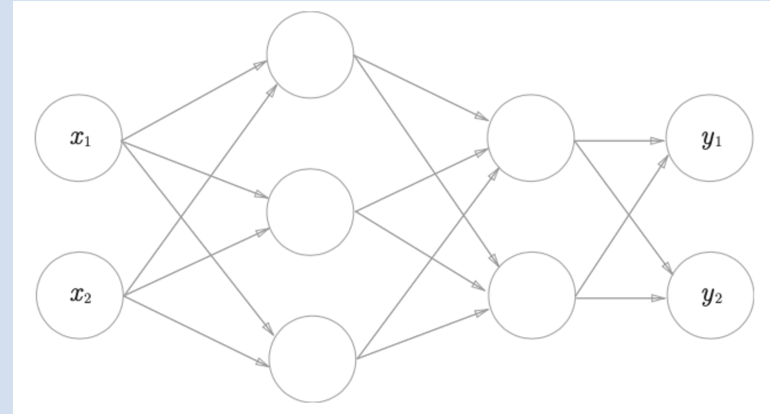
```
[0.3 0.7 1.1]
```

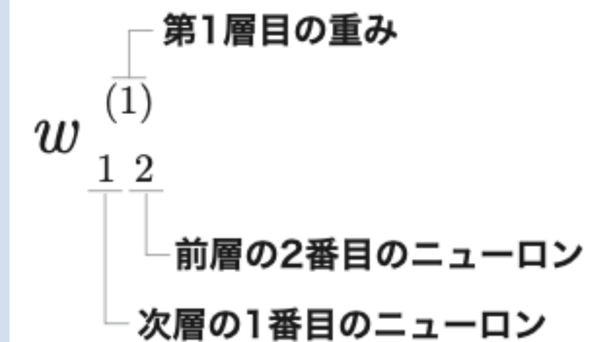
ニューラルネットワークの実装

■ 3層ニューラルネットワーク

② 第 1 層目に活性化関数を入れる







シグモイド関数を活性化関数として選ぶと、、、

$$\mathbf{A}^{(1)} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{B}^{(1)}$$

$$\mathrm{Z}^{(1)} = \underline{h(\mathrm{A}^{(1)})}$$ シグモイド関数

Pythonの実装例（neuralnet_1layer_sig.py）

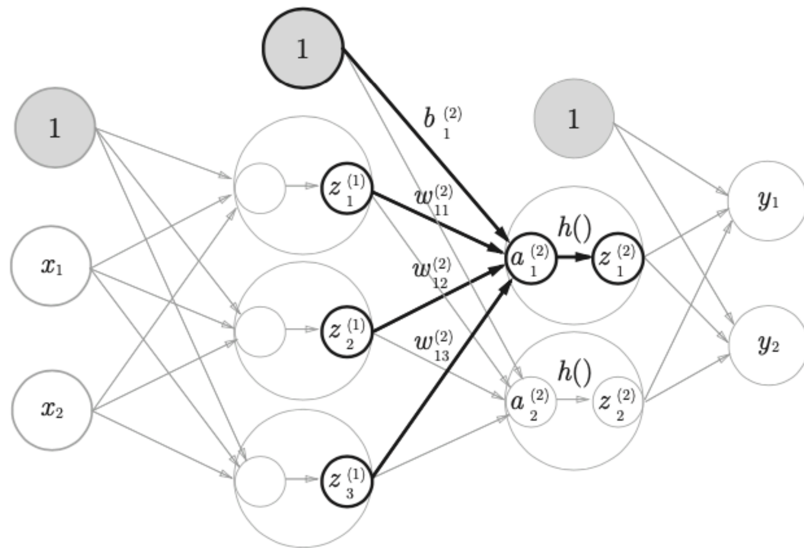
```
import numpy as np

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

X = np.array([1.0, 0.5])
W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)
print(Z1)
```

実行 →

```
[0.57444252 0.66818777 0.75026011]
```

ニューラルネットワークの実装

- 3層ニューラルネットワーク
  - ③ 第１層目から第２層目までの実装



Pythonの実装例（neuralnet_2layer.py）

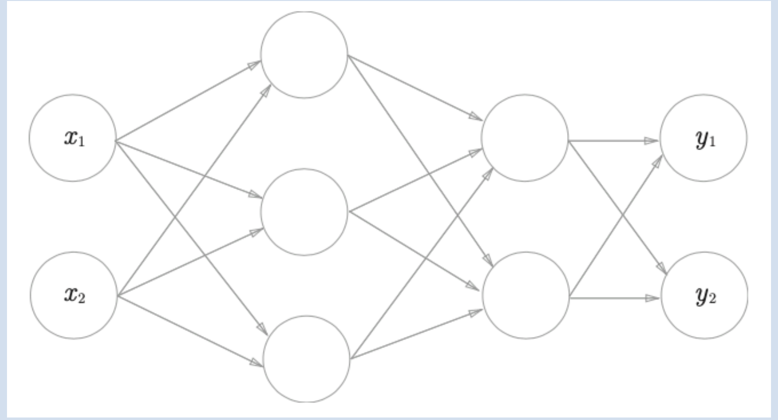
```
import numpy as np

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

X = np.array([1.0, 0.5])

W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])

W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)
print(Z1)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)
print(Z2)
```

実行

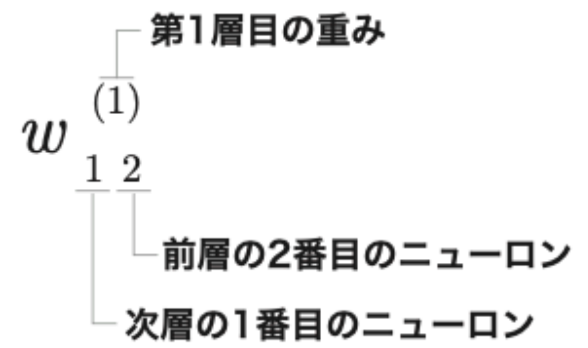
```
[0.57444252 0.66818777 0.75026011]
[0.62624937 0.7710107 ]
```

W2, B2, A2, Z2 を追加しただけ

3層ニューラルネットワーク



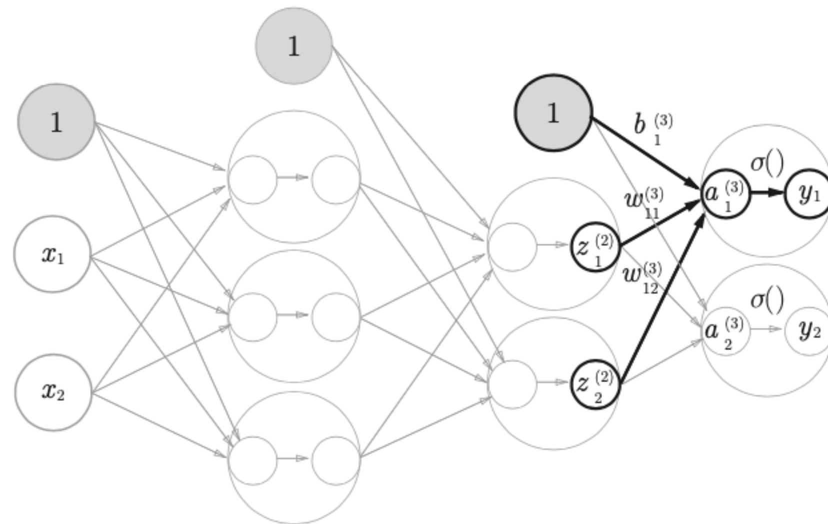
添字の見方



ニューラルネットワークの実装

■ 3層ニューラルネットワーク

③ 第 2 層目から第 3 層目までの実装



Pythonの実装例（neuralnet_3layer.py）

```
import numpy as np

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

def identity_function(x):
    return x

X = np.array([1.0, 0.5])

W1 = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
B1 = np.array([0.1, 0.2, 0.3])
W2 = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
B2 = np.array([0.1, 0.2])

W3 = np.array([[0.1, 0.3], [0.2, 0.4]])
B3 = np.array([0.1, 0.2])

A1 = np.dot(X, W1) + B1
Z1 = sigmoid(A1)

A2 = np.dot(Z1, W2) + B2
Z2 = sigmoid(A2)

A3 = np.dot(Z2, W3) + B3
Y = identity_function(A3)
print(Y)
```

W2, B2, A2 を追加して, 形を統一するため

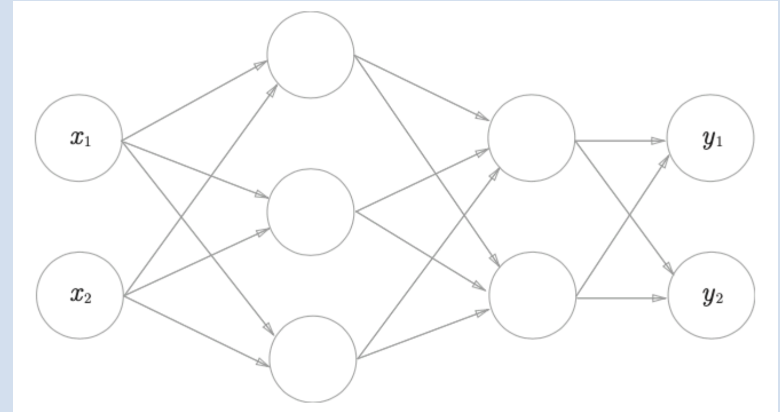
identity_function() - 恒等関数 に入れて, 出力 Y を得る形にしている

実行 →

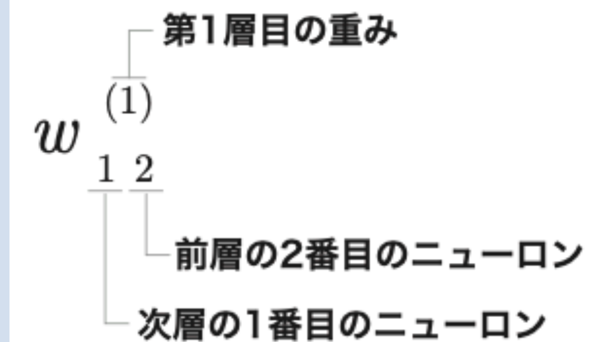
```
[0.31682708 0.69627909]
```

出力層での活性化関数は、
解く問題の性質に応じて決めていく
回帰問題 →恒等関数
2分類問題 →シグモイド関数
多クラス分類 →ソフトマックス関数

3層ニューラルネットワーク



添字の見方



ニューラルネットワークの実装

■ 3層ニューラルネットワーク

まとめ

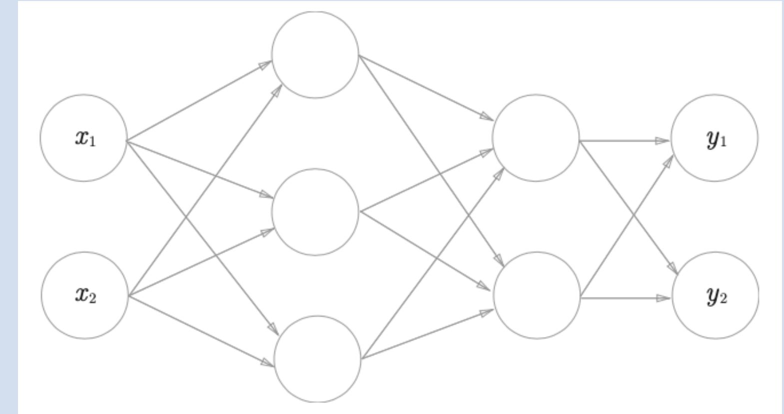
Pythonの実装例（neuralnet_3layer_matome.py）

```python
import numpy as np

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

def identity_function(x):
    return x

def init_network():
    network = {}
    network['W1'] = np.array([[0.1, 0.3, 0.5], [0.2, 0.4, 0.6]])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['W2'] = np.array([[0.1, 0.4], [0.2, 0.5], [0.3, 0.6]])
    network['b2'] = np.array([0.1, 0.2])
    network['W3'] = np.array([[0.1, 0.3], [0.2, 0.4]])
    network['b3'] = np.array([0.1, 0.2])

    return network

def forward(network, x):
    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']
    a1 = np.dot(x, W1) + b1
    z1 = sigmoid(a1)
    a2 = np.dot(z1, W2) + b2
    z2 = sigmoid(a2)
    a3 = np.dot(z2, W3) + b3
    y = identity_function(a3)

    return y

network = init_network()
x = np.array([1.0, 0.5])
y = forward(network, x)
print(y)
```

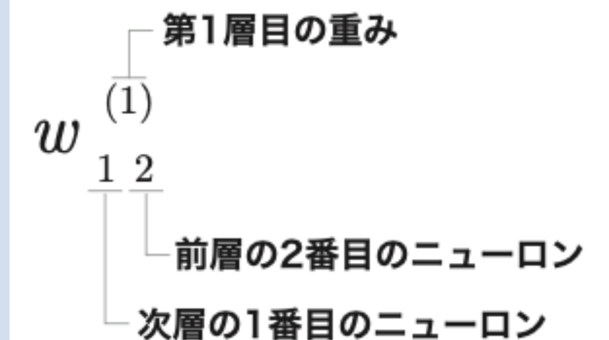
実行 →

```
[0.31682708 0.69627909]
```

3層ニューラルネットワーク



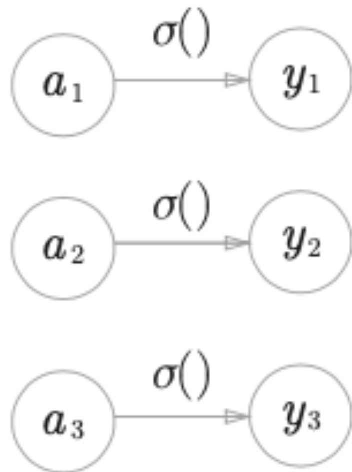
添字の見方



入力→出力（フォワード）方向の実装完了！

出力層

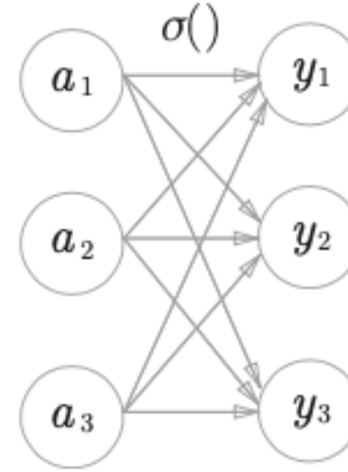
■ 恒等関数



入力がそのまま
出力になる

回帰問題で一般的に使われる

■ ソフトマックス関数



出力のニューロンが
全ての入力の影響を
うける！

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

Pythonの実装例（softmax.py）

```
import numpy as np

def softmax(x):
    exp_x = np.exp(x)
    sum_exp_x = np.sum(exp_x)
    return exp_x / sum_exp_x

x = np.array([0.3, 2.9, 4.0])
y = softmax(x)
print(y)
```

実行 →

```
[0.01821127 0.24519181 0.73659691]
```

1.8%　24.5%　73.7%

＜特徴＞
- 出力を確率として解釈できる
- 入出力の要素の大小関係は変わらない

分類問題で一般的に使われる

→ **なぜソフトマックス関数を出力層に使うのかは学習の話の時に**

出力層：ソフトマックス関数

- ソフトマックス関数の実装上の注意
  - expの肩に 10^3 桁の値がのるとオーバーフローしてしまう

＜対策＞

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

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

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

→ expの肩に足したり、
引いたりしても結果は変わらない！

Pythonの実装例（softmax_overflow.py）

```
import numpy as np

def softmax(x):
    c = np.max(x)
    exp_x = np.exp(x-c)
    sum_exp_x = np.sum(exp_x)
    return exp_x / sum_exp_x

x = np.array([1010, 1000, 990])
y = softmax(x)
print(y)
```

max関数を使う

手書き数字認識

- 使用するデータセット
  - MNISTデータセット：0 ~ 9 の数字画像からなるデータセット

＜中身の確認＞

```
$ cd ./ch03
$ python3 mnist_show.py
```

- ニューラルネットワークのセットアップ
  - 入力層：28 x 28 = 784個（画像サイズ）
  - 出力層：10個（0 ~ 9 の数字）
  - 隠れ層：2層
  - ニューロン：1層目 50個、2層目 100個
  - 重み：すでに学習された sample_weight.pkl を使用する

＜実行＞

```
$ python3 neuralnet_mnist.py
```

手書き数字認識

- 数字認識処理の流れ

1. MNISTのデータを取得（x, t = get_data()）
2. for文で順に画像データを predict() に入れ、分類を行う

   隠れ層：sigmoid関数

   出力層：softmax関数
3. 最も確率の高い要素のインデックスを取得し、予測結果とする
4. ニューラル ネットワークが予測した答えと正解ラベルとを比較
5. 正解した割合（**認識精度**）を出す

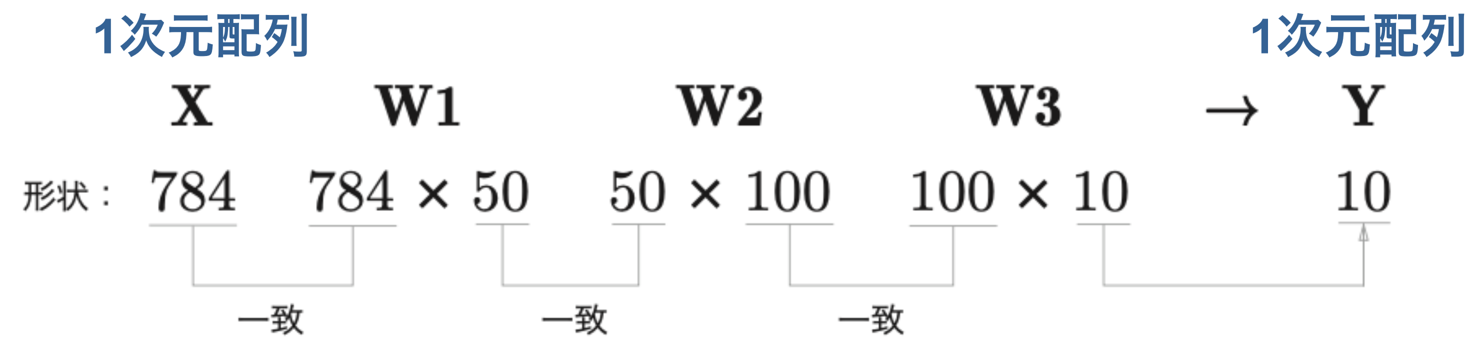
※今回は、データの値が0.0~1.0の範囲に収まるよう**正規化**している

入力データに前もって変換を行うことを**前処理**という

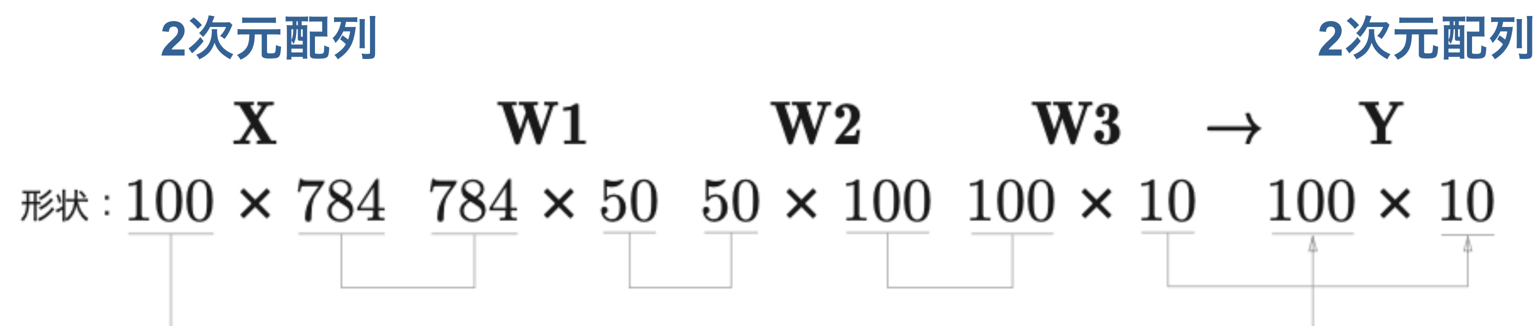
前処理はdeep learningの学習の高速化のためなどで使われる

バッジ処理

- バッチ処理
  - 入力データを束（**Batch**）にして処理すること
  - 各層の重みの形状



  - バッチ処理における各層の重みの形状



<実行>

```
$ python3 neuralnet_mnist_batch.py
```

バッジ処理

■ バッチ処理

- neuralnet_mnist_batch

```
x, t = get_data()
network = init_network()

batch_size = 100 # バッチの数
accuracy_cnt = 0

for i in range(0, len(x), batch_size):
    x_batch = x[i:i+batch_size]
    y_batch = predict(network, x_batch)
    p = np.argmax(y_batch, axis=1)
    accuracy_cnt += np.sum(p == t[i:i+batch_size])

print("Accuracy:" + str(float(accuracy_cnt) / len(x)))
```

- **x[i:i+batch_size]**
  先頭から 100 枚ずつバッチとして取り出す
- **np.argmax(y_batch, axis=1)**
  100 × 10 の配列 の中で 1 次元目の要素ごとに最大値のインデックスを見つ ける
- **p == t[i:i+batch_size]**
  バッチ単位で分類した結果と
  実際の答えを比較

バッチ処理を行うことで、高速に効率良く処理することができる！

まとめ

- ニューラルネットワークでは、活性化関数としてシグモイド関数や ReLU 関数のような滑らかに変化する関数を利用する。
- NumPy の多次元配列をうまく使うことで、ニューラルネットワークを効 率良く実装することができる。
- 機械学習の問題は、回帰問題と分類問題に大別できる。
- 出力層で使用する活性化関数は、回帰問題では恒等関数、分類問題ではソフトマックス関数を一般的に利用する。
- 分類問題では、出力層のニューロンの数を分類するクラス数に設定する。
- 入力データのまとまりをバッチと言い、バッチ単位で推論処理を行うことで、計算を高速に行うことができる。

Backup