ニューラルネット(1)

前田利之（まえだ　としゆき）
maechan@hannan-u.ac.jp

阪南大学　経営情報学部

(2024.10.25)

先週の自由課題の解答例

☆ 以下を mymodules の中にhadder_gate.py という名前でファイルに保存する（念のため、全ゲート入りのコードへのリンクがはってあるので、右クリックで保存して利用も可）

```
# coding: utf-8
import numpy as np
from and_gate import AND
from xor_gate import XOR
def HADDER(x1, x2):
    c = AND(x1, x2)
    s = XOR(x1, x2)
    return c,s # 2値を一度に返せる！
```

先週の自由課題の解答例

☆ 以下を Colab. 上で実行する（保存は hadder.ipynb という名前で）

```
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)
from hadder_gate import HADDER
for xs in [(0, 0), (1, 0), (0, 1), (1, 1)]:
    c,s = HADDER(xs[0], xs[1]) #２値を受け取れる
    print(str(xs)+" -> "+str(c)+' '+str(s))
```

ニューラルネットワークとは（再掲）

- 人間の脳の中には［　　］という神経細胞がたくさんある（千億個のオーダー）
- 各［　　］が［　　］と呼ばれる接合部位によって繋がっている
- ［　　］は入力される電気信号の合計が、ある一定の量（閾値）を超えると［　　］し、シナプスによって次のニューロンに電気信号を出力する
- この動作の連続により、脳は信号の伝達を行っている

ニューラルネットワークとは（再掲）

- 人間の脳の中には ニューロン という神経細胞がたくさんある（千億個のオーダー）
- 各 [　　] が [　　] と呼ばれる接合部位によって繋がっている
- [　　] は入力される電気信号の合計が、ある一定の量（閾値）を超えると [　　] し、シナプスによって次のニューロンに電気信号を出力する
- この動作の連続により、脳は信号の伝達を行っている

ニューラルネットワークとは（再掲）

- 人間の脳の中には ニューロン という神経細胞がたくさんある（千億個のオーダー）
- 各 ニューロン が [　　] と呼ばれる接合部位によって繋がっている
- [　　] は入力される電気信号の合計が、ある一定の量（閾値）を超えると [　　] し、シナプスによって次のニューロンに電気信号を出力する
- この動作の連続により、脳は信号の伝達を行っている

ニューラルネットワークとは（再掲）

- 人間の脳の中には ニューロン という神経細胞がたくさんある（千億個のオーダー）
- 各 ニューロン が シナプス と呼ばれる接合部位によって繋がっている
- [　　] は入力される電気信号の合計が、ある一定の量（閾値）を超えると [　　] し、シナプスによって次のニューロンに電気信号を出力する
- この動作の連続により、脳は信号の伝達を行っている

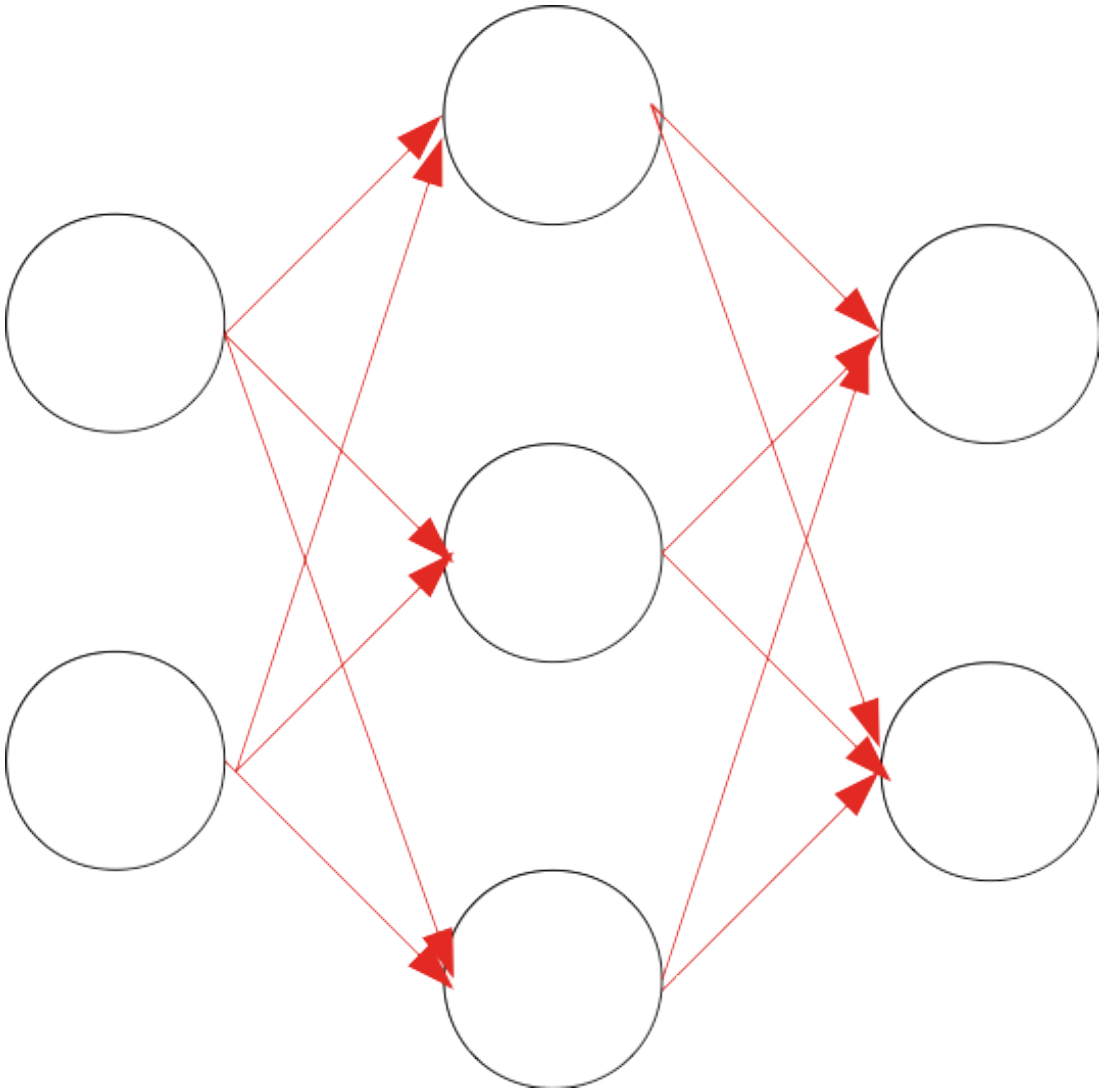
ニューラルネットワークとは（再掲）

- 人間の脳の中には ニューロン という神経細胞がたくさんある（千億個のオーダー）
- 各 ニューロン が シナプス と呼ばれる接合部位によって繋がっている
- ニューロン は入力される電気信号の合計が、ある一定の量（閾値）を超えると [　　] し、シナプスによって次のニューロンに電気信号を出力する
- この動作の連続により、脳は信号の伝達を行っている

ニューラルネットワークとは（再掲）

- 人間の脳の中には **ニューロン** という神経細胞がたくさんある（千億個のオーダー）
- 各 **ニューロン** が **シナプス** と呼ばれる接合部位によって繋がっている
- **ニューロン** は入力される電気信号の合計が、ある一定の量（閾値）を超えると **発火** し、シナプスによって次のニューロンに電気信号を出力する
- この動作の連続により、脳は信号の伝達を行っている

ニューラルネットワークの例

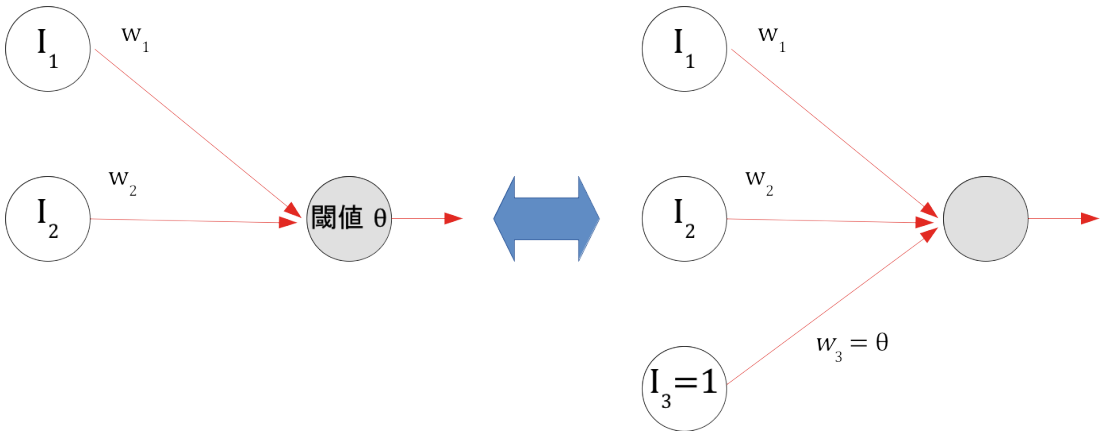


活性化関数

☆２入力のパーセプトロンの出力式
(y:出力, b:バイアス（閾値）, w_1, w_2: 重み, I_1, I_2:入力)

$$y = \begin{cases} 1 & (b + w_1 I_1 + w_2 I_2 > 0) \\ 0 & (otherwise) \end{cases}$$

活性化関数（イメージ）



☆b(バイアス) が w_3 であり、 I_3 は常に１となっている

活性化関数

☆ h() という関数を導入し、前式を書き換える

$$y = h(b + w_1 I_1 + w_2 I_2)$$

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

$y = h(b + w_1 I_1 + w_2 I_2)$ は、

$$a = b + w_1 I_1 + w_2 I_2$$

$$y = h(a)$$

とも書ける。この h() こそが [　　]

活性化関数

☆ h() という関数を導入し、前式を書き換える

$$y = h(b + w_1 I_1 + w_2 I_2)$$

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

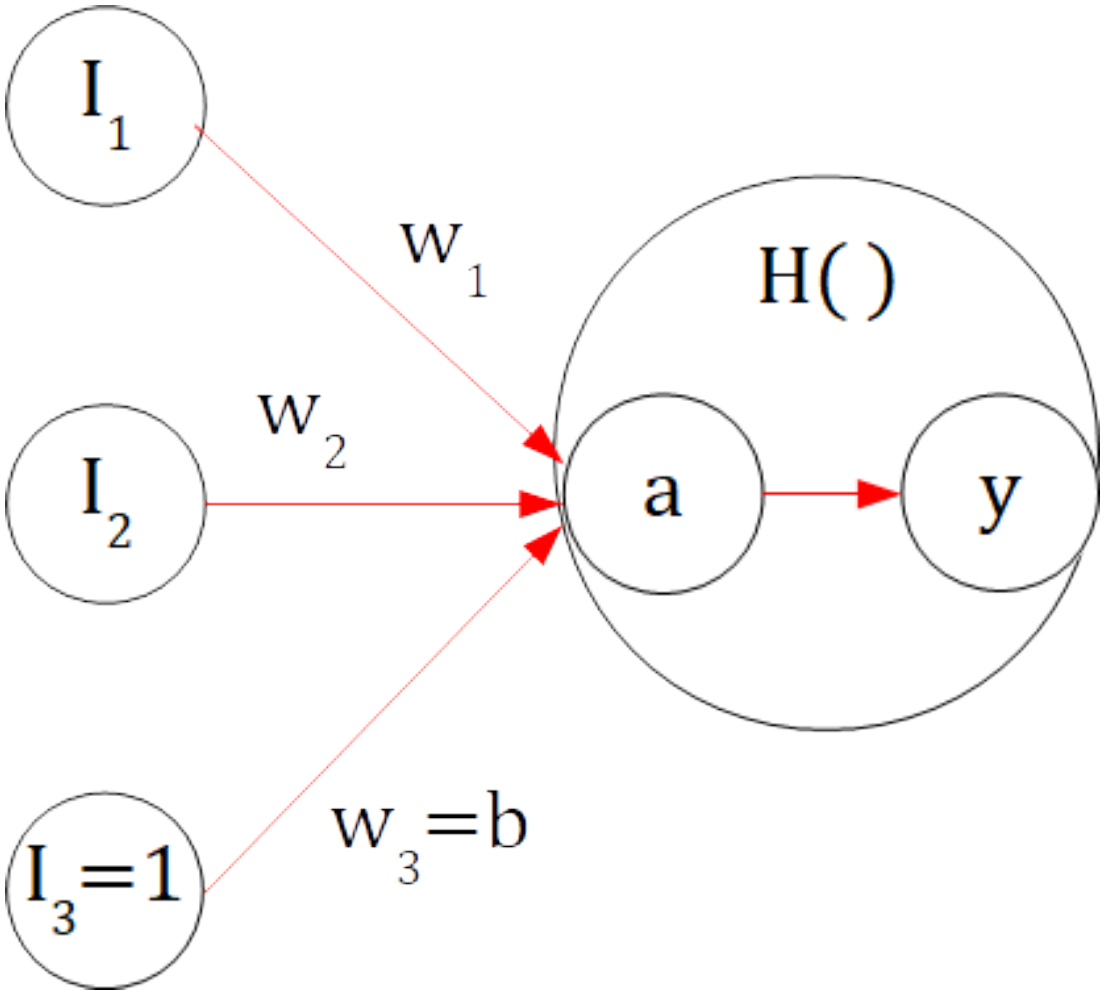
$y = h(b + w_1 I_1 + w_2 I_2)$ は、

$$a = b + w_1 I_1 + w_2 I_2$$

$$y = h(a)$$

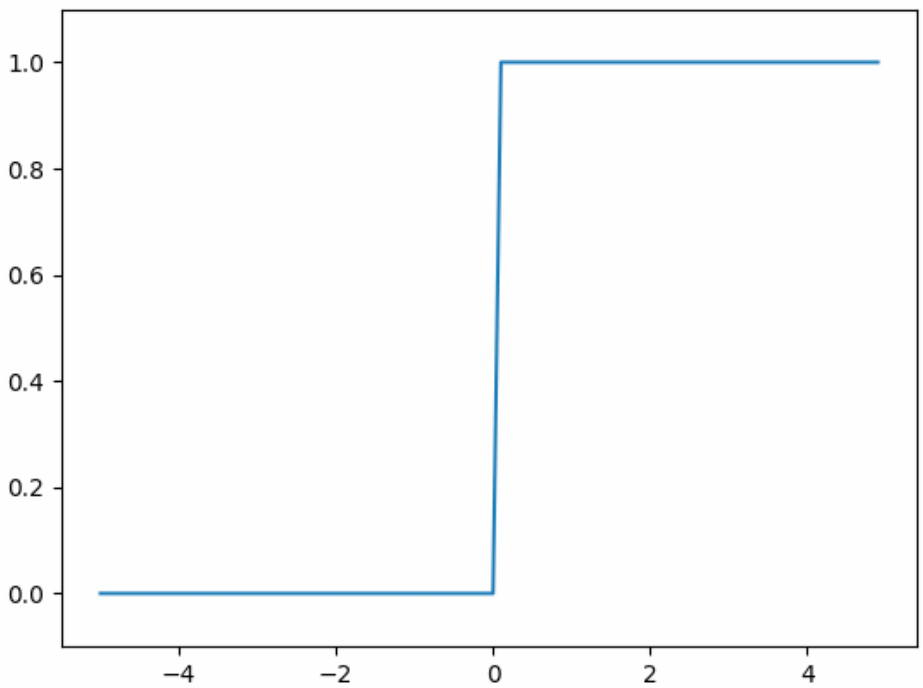
とも書ける。この h() こそが **活性化関数**

活性化関数



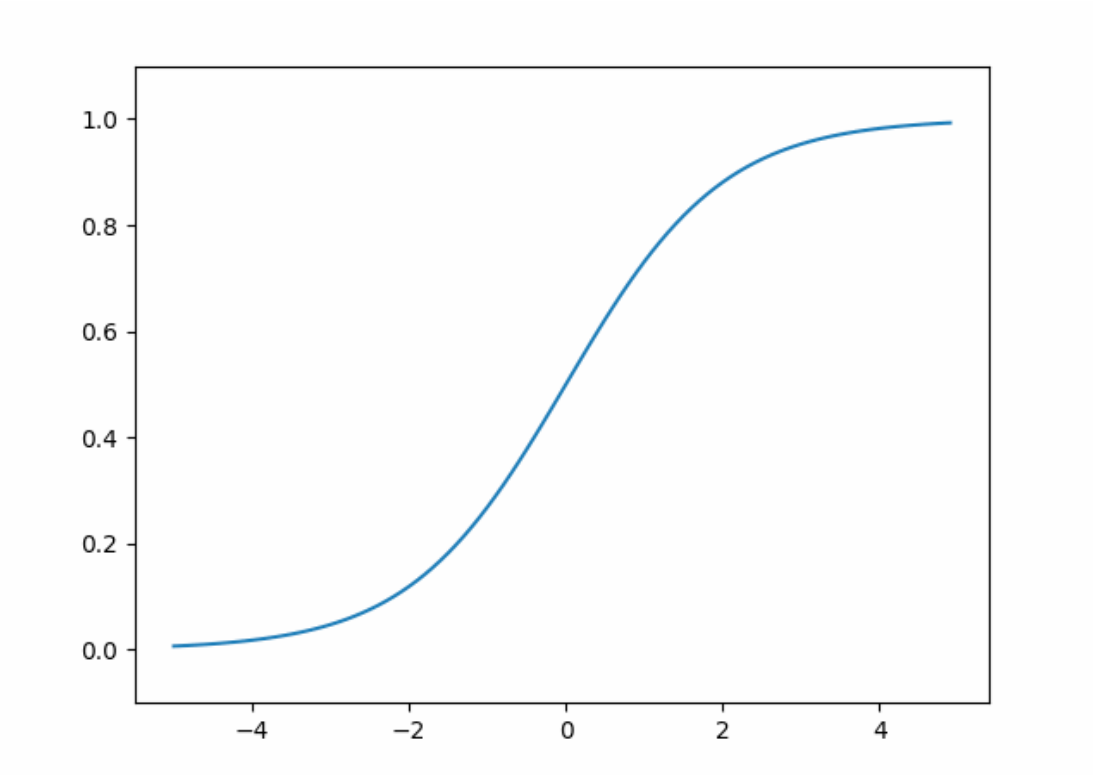
ステップ関数

- 前頁の h() 【階段関数】…実は、これだけではない（後述）



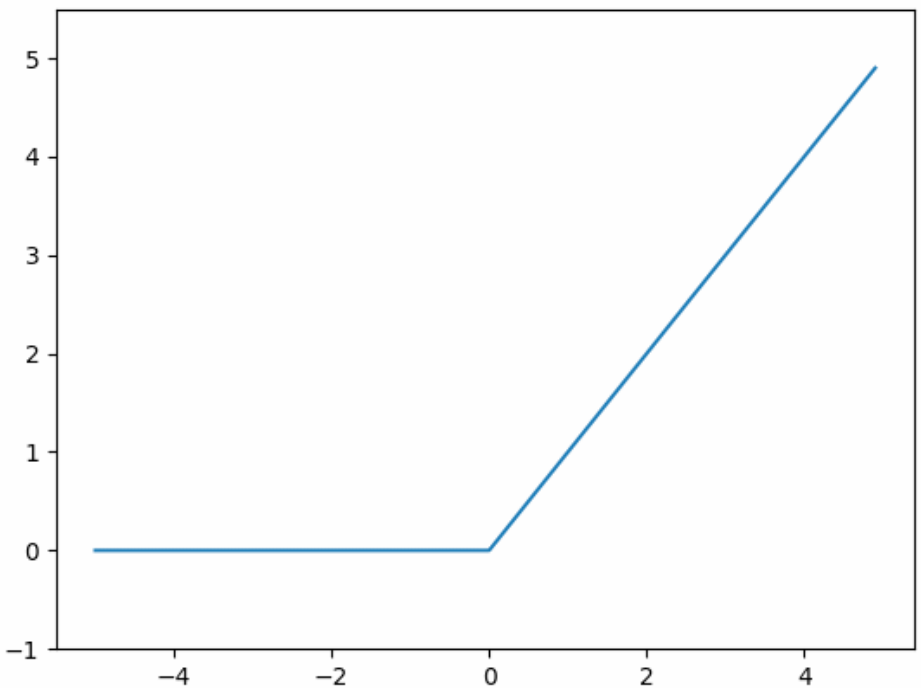
シグモイド関数

- 活性化関数の代表



ReLU

- Rectified Linear Unit
- 最近よく使われる活性化関数



ステップ・シグモイド・ReLU関数の実装

☆以下を activ.py という名前のファイルに保存してから Colab に貼り付け実行する（次頁へ続く）

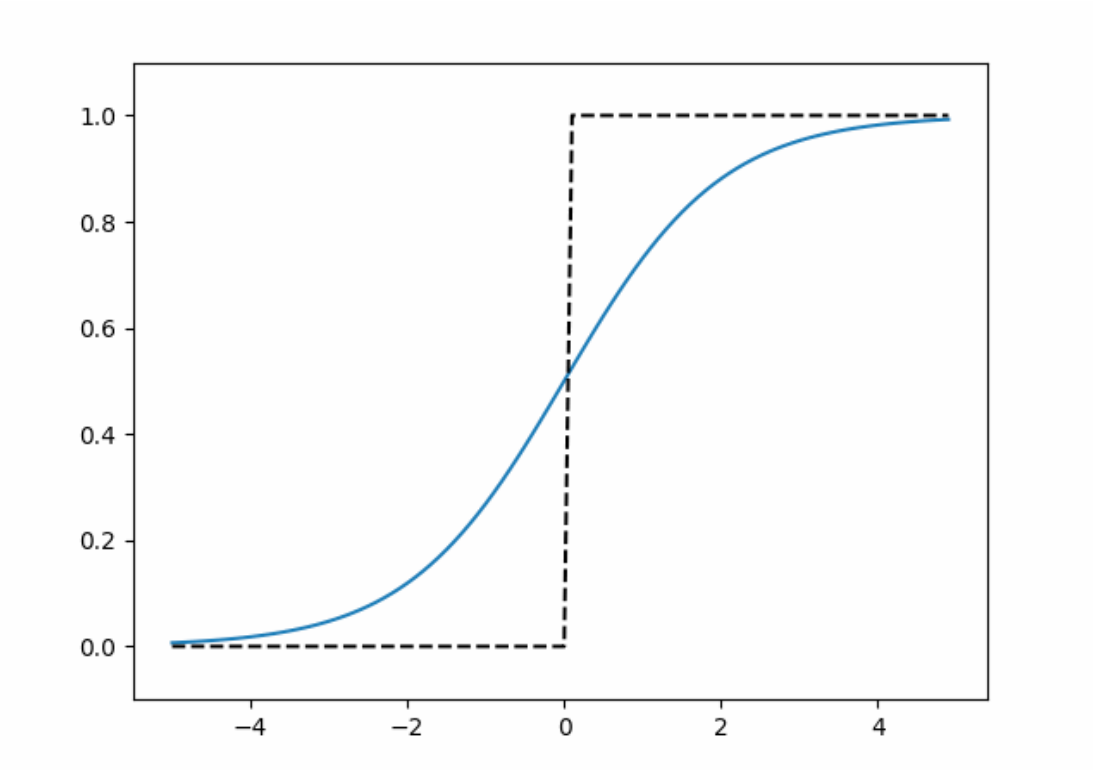
```
import numpy as np
import matplotlib.pylab as plt
def step_function(x):
    return np.array(x > 0, dtype=np.int)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def reru(x):
    return np.maximum(0,x)
```

ステップ・シグモイド・ReLU関数の実装

（前頁からの続き）

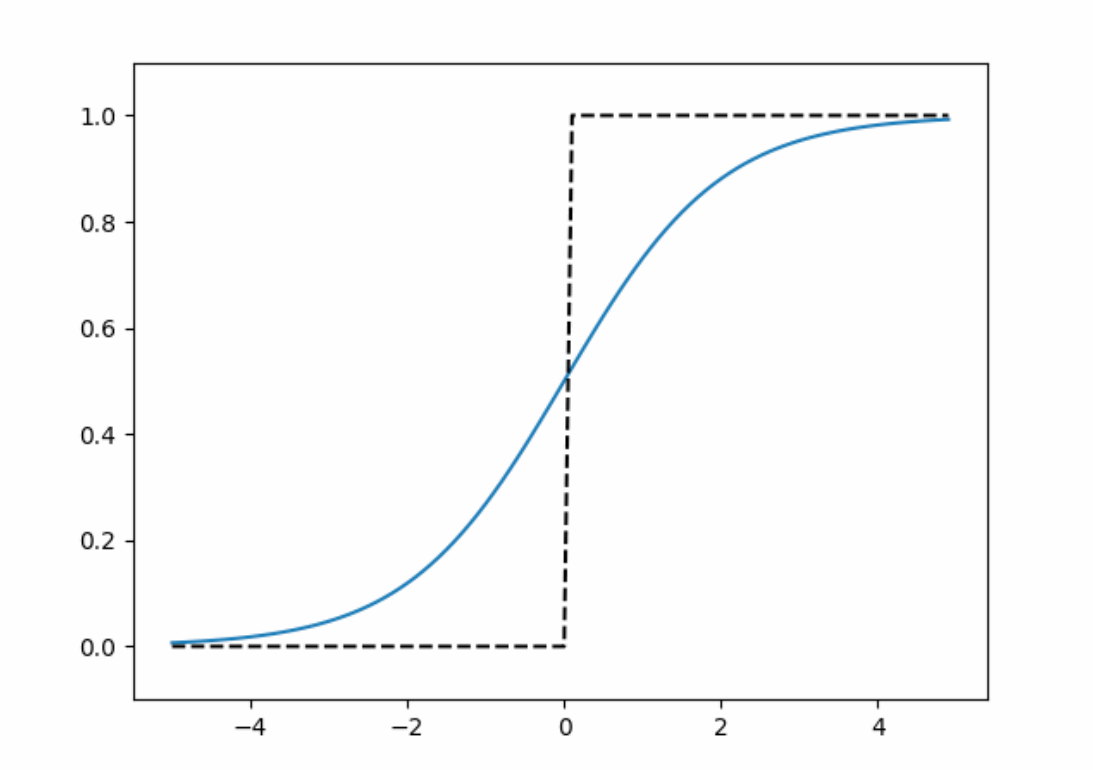
```
# -5〜5 まで 0.1 刻みの配列を生成 (X 軸)
X = np.arange(-5.0, 5.0, 0.1)
Y = sigmoid(X)
# ↑を step_function(X) か relu(X) に
# 置き換えるとそのグラフを描画
plt.plot(X, Y)
# 図で描画する y 軸の範囲を指定
plt.ylim(-0.1, 1.1)
plt.show()
```

ステップ関数とシグモイド関数の比較



☆ シグモイド関数は [　　] （実は微分可能←これが大事）(微分については別資料（AI・データサイエンス基礎／先端技術論）参照)

ステップ関数とシグモイド関数の比較



☆ シグモイド関数は **滑らか** （実は微分可能←これが大事）(微分については別資料（AI・データサイエンス基礎／先端技術論）参照)

非線形関数

- これまでのステップ・シグモイド・ReLU 関数は全て非線形関数
  - 線形とは何か、というのは別資料（旧・先端技術論）の資料を参照のこと
- 線形だと、多層化する意味がなくなる（証明は略）

準備：多次元配列

- そもそも多次元配列とは：　　の
  　
- １次元（１列）、２次元（行列）、…
- ニューラルネットの実装には numpy が便利= 多次元配列を簡単に扱える

準備：多次元配列

- そもそも多次元配列とは： 数（値） の
- １次元（１列)、２次元（行列)、…
- ニューラルネットの実装には numpy が便利= 多次元配列を簡単に扱える

準備：多次元配列

- そもそも多次元配列とは：**数（値）**の**あつまり（集合）**
- １次元（１列）、２次元（行列）、…
- ニューラルネットの実装には numpy が便利= 多次元配列を簡単に扱える

多次元配列

☆ 以下を Colab 上で実行する

```
# coding: utf-8
import numpy as np
A = np.array([1,2,3,4])
print(np.ndim(A))
print(A.shape)
print(A.shape[0])
B = np.array([[1,2],[3,4],[5,6]])
print(B)
print(np.ndim(B))
print(B.shape)
```

準備：多次元配列

☆ 実行結果

```
1
(4,)
4
[[1 2]
 [3 4]
 [5 6]]
2
(3, 2)
```

ニューラルネットワークとは（続き）

☆前頁の出力行の意味は、それぞれ：

- 「１次元」
- 「要素が４」（括弧付きなのは多次元と統一した記述）
- 「要素４」
- 行列としての、中身を表示
- 「２次元」
- 「３行２列」

行列の積

※行列の積（線形代数の復習）

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = A, \begin{pmatrix} e & f \\ g & h \end{pmatrix} = B$$

とすると、

$$AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

※左項の行と右項の列をそれぞれ掛けたもので構成される行列

行列の積（続き）

$$AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

であったが、ちなみに

$$BA = \begin{pmatrix} ae + cf & be + df \\ ag + ch & bg + dh \end{pmatrix}$$

となるので、一般に $AB = BA$ である

※要するに、勝手に順番を変えてはいけないということ（重要！）

行列の積（続き）

$$AB = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

であったが、ちなみに

$$BA = \begin{pmatrix} ae + cf & be + df \\ ag + ch & bg + dh \end{pmatrix}$$

となるので、一般に $\boxed{AB \neq BA}$ である

※要するに、勝手に順番を変えてはいけないということ（重要！）

行列の積（実習）

☆ 以下を Colab 上で実行する

```
A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])
print(np.dot(A,B))
```

☆ 実行結果

```
[[19 22]
 [43 50]]
```

行列の積

※正方行列（縦横一緒）でなくても、

していれば積は計算できる

$$\begin{pmatrix} g & h & i \\ j & k & l \end{pmatrix} = A$$

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} = B,$$

とすると、

$$AB = \begin{pmatrix} ag + ch + ei & bg + dh + fi \\ aj + ck + el & bj + dk + fl \end{pmatrix}$$

行列の積

※正方行列（縦横一緒）でなくても、
左項の列数と右項の行数が一致していれば積は計算できる

$$\begin{pmatrix} g & h & i \\ j & k & l \end{pmatrix} = A$$

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} = B,$$

とすると、

$$AB = \begin{pmatrix} ag + ch + ei & bg + dh + fi \\ aj + ck + el & bj + dk + fl \end{pmatrix}$$

行列の積（実習）

☆ 以下を Colab 上で実行する

```
C = np.array([[1,2,3],[4,5,6]])
D = np.array([[1,2],[3,4],[5,6]])
print(np.dot(C,D))
print(np.dot(D,C))
```

準備：多次元配列

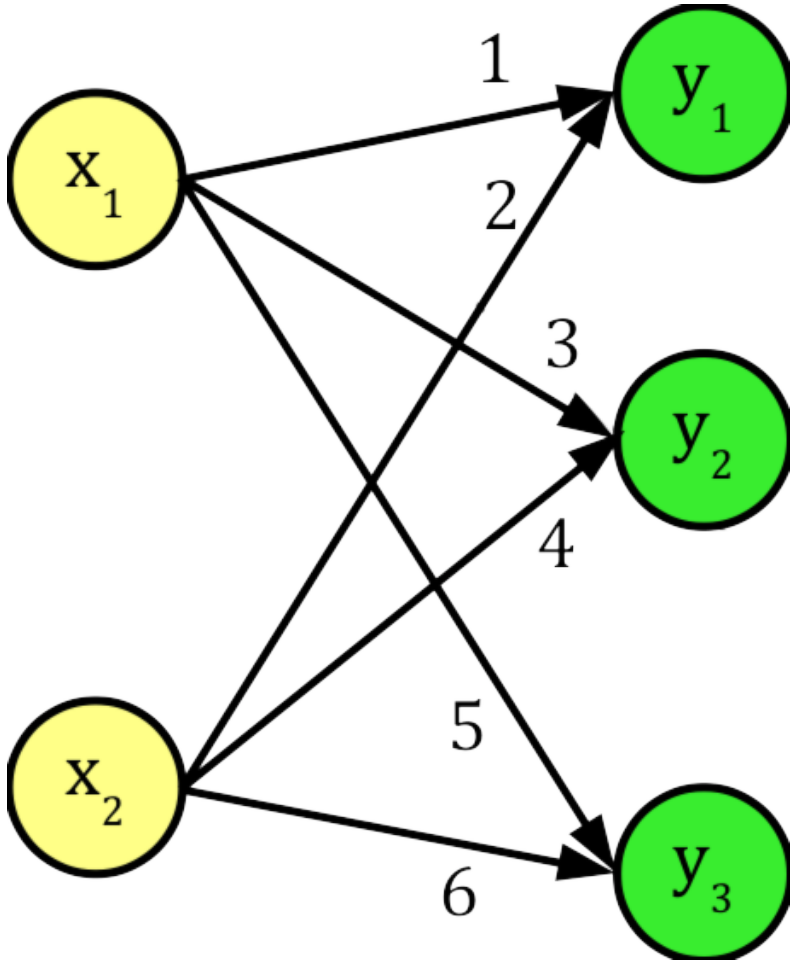
☆ 実行結果

```
[[22 28]
 [49 64]]
[[ 9 12 15]
 [19 26 33]
 [29 40 51]]
```

- 順序を逆にすると行列の サイズも変わることに注意！

ニューラルネットの行列の積

☆ 下のようなニューラルネットを考える



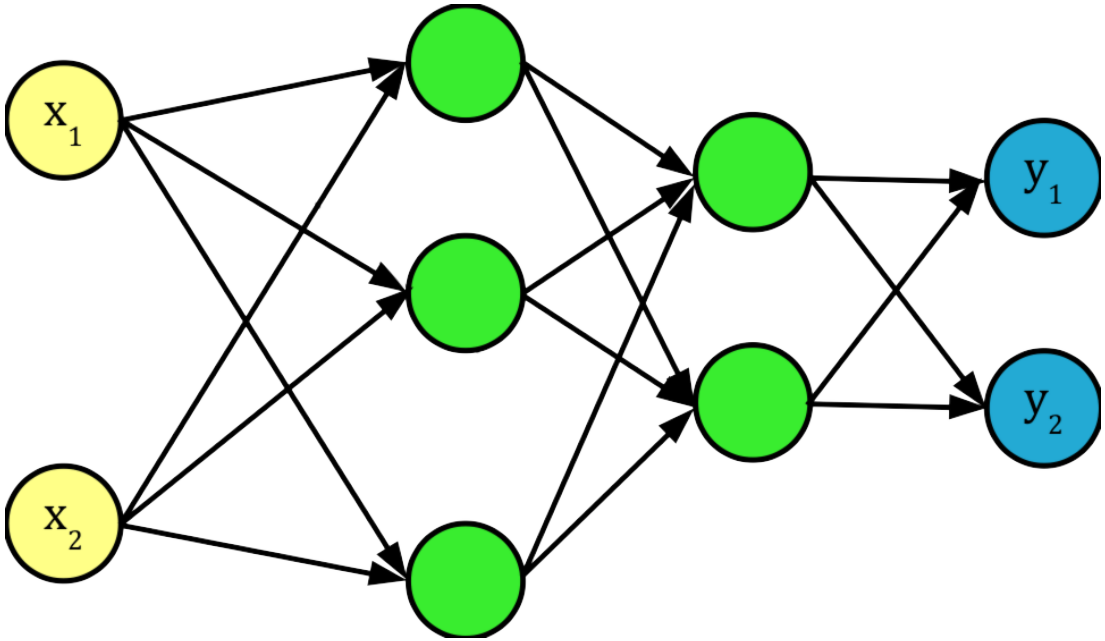
ニューラルネットの行列の積

☆ 前頁で、入力が $(x_1, x_2) = (1, 2)$ のときの出力は以下で計算できるので Colab 上で確かめる

```
X = np.array([1,2])
W = np.array([[1,3,5],[2,4,6]])
print(W.shape) # ２行３列を確かめる
Y = np.dot(X,W)
print(Y)
```

ニューラルネットの実装

☆ 以降、３層ニューラルネットを考える



ニューラルネットの実装（記号について）

☆ 前頁の３層ニューラルネットの個々について、以下の記号を用いる

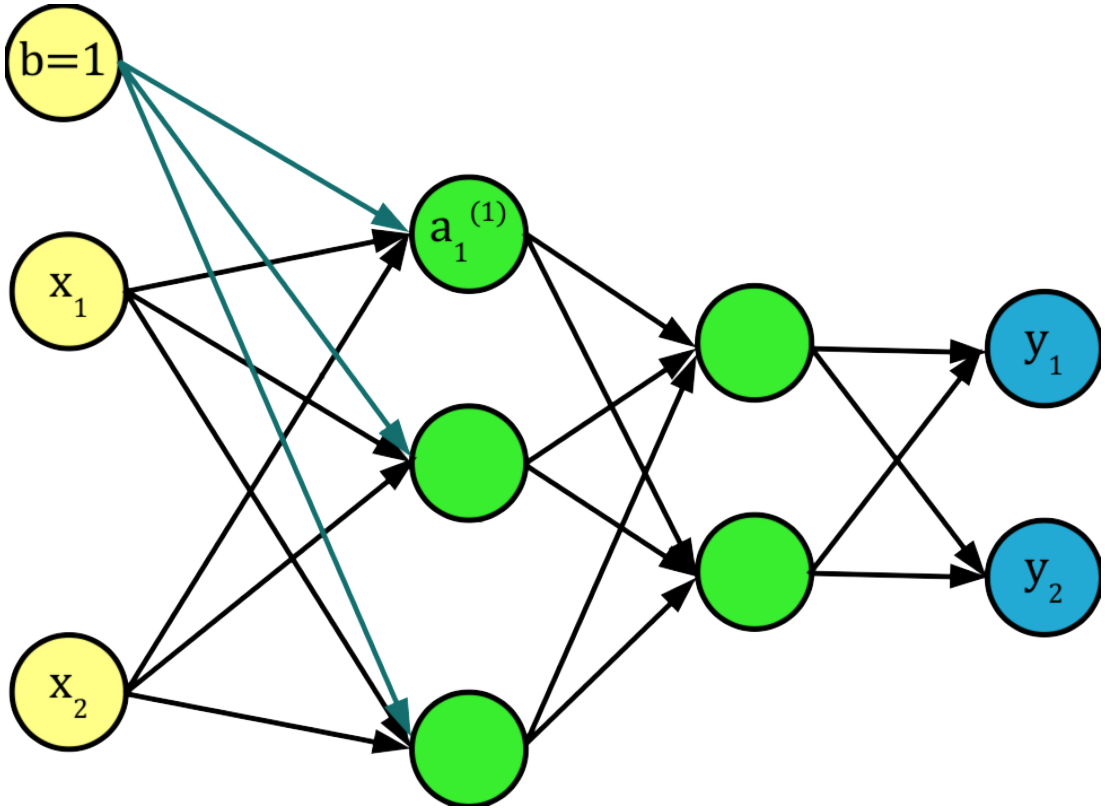
- $a_2^{(1)}$: (1) 層目の２番目のニューロン
- $w_{13}^{(2)}$: 前層の３番目のニューロンから、(2) 層目の１番目のニューロンへの重み [次層の番号、前層の番号の順であることに注意！]

ニューラルネットの実装

☆ $a_1^{(1)}$ を式であらわすと、

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

ニューラルネットの実装



ニューラルネットの実装

☆行列の積を用いると、１層目全部の
[　　]は以下であらわされる

$$A^{(1)} = XW^{(1)} + B^{(1)} \tag{2}$$

但し、

$$A^{(1)} = (a_1^{(1)}, a_2^{(1)}, a_3^{(1)}),$$
$$X = (x_1, x_2),$$
$$B^{(1)} = (b_1^{(1)}, b_2^{(1)}, b_3^{(1)}),$$

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

ニューラルネットの実装

☆行列の積を用いると、１層目全部の
「重み付き和」は以下であらわされる

$$A^{(1)} = XW^{(1)} + B^{(1)} \tag{2}$$

但し、

$$A^{(1)} = (a_1^{(1)}, a_2^{(1)}, a_3^{(1)}),$$
$$X = (x_1, x_2),$$
$$B^{(1)} = (b_1^{(1)}, b_2^{(1)}, b_3^{(1)}),$$

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

ニューラルネットの実装

☆ 以下を Colab 上で実行して確かめる

```
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)
```

◎結果は [0.3 0.7 1.1]

ニューラルネットの実装

☆ 活性化関数として前述の sigmoid() を用いる。前頁に引続き、以下を Colab 上で実行して確かめる

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
Z1 = sigmoid(A1)
print(Z1)
```

◎結果は [0.57444252, 0.66818777, 0.75026011]

ニューラルネットの実装

☆ 同様に、２層目、３層目についても実装する。最終層の活性化関数のみ、これまでの層（隠れ層）とは違うことに注意して、前頁に引続き、以下を Colab 上で実行して確かめる。

```
W2 = np.array([[0.1,0.4],[0.2,0.5],[0.3,0.6]])
B2 = np.array([0.1,0.2])
A2 = np.dot(Z1,W2)+B2
Z2 = sigmoid(A2)
def identify_function(x):
  return x
W3 = np.array([[0.1,0.3],[0.2,0.4]])
B3 = np.array([0.1,0.2])
A3 = np.dot(Z2,W3)+B3
Y = identify_function(A3)
```

ニューラルネットの実装

- 前頁の結果は
- 最後に使っている identify_function() は恒等関数（そのまま返す）で、無くても良いが、活性化関数を通すという流れを統一するために導入している
  - ここまでのコードをまとめて、関数などを整理しなおしたものを、3layer-nn.py に置いてあるので、右クリックでダウンロードして、動かしてみてから中身を確認してください！

ニューラルネットの実装

- 前頁の結果は [0.31682708 0.69627909]
- 最後に使っている identify_function() は恒等関数（そのまま返す）で、無くても良いが、活性化関数を通すという流れを統一するために導入している
  - ここまでのコードをまとめて、関数などを整理しなおしたものを、3layer-nn.py に置いてあるので、右クリックでダウンロードして、動かしてみてから中身を確認してください！

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！