

## ②活性化関数\_実装演習

=====

目次: (なお、本実装演習のコードは深層学習\_前編day1のSection1~3までの内容を横断的に含んだものである)

### 0.中間層の活性化関数の実装

#### 1.順伝播 (単層・単ユニット)

#### 2.順伝播 (単層・複数ユニット)

#### 3.順伝播 (3層・複数ユニット)

#### 4.多クラス分類 (2-3-4ネットワーク)

#### 5.回帰 (2-3-2ネットワーク)

#### 6.2値分類 (2-3-1ネットワーク)

→順伝播ニューラルネットワークによる、入力層~中間層・出力層、活性化関数の実装方法を学習

### 【要約】

- ・ 中間層の活性化関数:シグモイド関数、ReLU関数を使用(一般的なニューラルネットワークの場合)

ステップ関数(パーセプトロンの場合など)

- ・ 出力層の活性化関数:取り扱う問題により異なる

→多クラス分類:ソフトマックス関数 複数候補の中から、確率論的に答えを導きたいため

→回帰:恒等関数 予測結果をそのまま出力したいため

→2値分類:シグモイド関数 2者の比較のため、0 or 1のどちらに依っているか分かれば十分なため

In [4]:

```
import sys
sys.path.append('C:/Users/NIF/Desktop/(削除)rabbitt/DNN_code_colab_lesson_1_2/DNN_code_colab_lesson_1_2')

import numpy as np
from common import functions

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    #print("shape: " + str(x.shape))
    print("")
```

=====

## 0.中間層の活性化関数の実装

In [1]:

```
import numpy as np

# 中間層の活性化関数
# シグモイド関数（ロジスティック関数）
def sigmoid(x):
    return 1/(1 + np.exp(-x))

# ReLU関数
def relu(x):
    return np.maximum(0, x)

# ステップ関数（閾値0）
def step_function(x):
    return np.where(x > 0, 1, 0)
```

→シグモイド関数 数式のとおり

→ReLU関数 0と引数(数値)を比較し、大きい方を出力[maximumにより]

0以下の場合は0、0より大きい場合はそのまま出力される

→ステップ関数 0以下は0、0より大きければ1を出力

=====

## 1.順伝播（単層・単ユニット）

In [5]:

```
# 順伝播（単層・単ユニット）

# 重み
W = np.array([[0.1], [0.2]])

## 試してみよう_配列の初期化
#W = np.zeros(2)
#W = np.ones(2)
#W = np.random.rand(2)
#W = np.random.randint(5, size=(2))

print_vec("重み", W)

# バイアス
b = 0.5

## 試してみよう_数値の初期化
#b = np.random.rand() # 0~1のランダム数値
#b = np.random.rand() * 10 -5 # -5~5のランダム数値

print_vec("バイアス", b)

# 入力値
x = np.array([2, 3])
print_vec("入力", x)

# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.relu(u)
print_vec("中間層出力", z)
```

\*\*\* 重み \*\*\*

```
[[0.1]
 [0.2]]
```

\*\*\* バイアス \*\*\*

```
0.5
```

\*\*\* 入力 \*\*\*

```
[2 3]
```

\*\*\* 総入力 \*\*\*

```
[1.3]
```

\*\*\* 中間層出力 \*\*\*

```
[1.3]
```

→総入力:中間層での計算値  $u = x * W + b$

→中間出力:中間層での計算値(上記)をReLU関数で活性化し出力

=====

## 2.順伝播（単層・複数ユニット）

In [6]:

```
# 順伝播（単層・複数ユニット）

# 重み
W = np.array([
    [0.1, 0.2, 0.3],
    [0.2, 0.3, 0.4],
    [0.3, 0.4, 0.5],
    [0.4, 0.5, 0.6]
])

## 試してみよう_配列の初期化
#W = np.zeros((4,3))
#W = np.ones((4,3))
#W = np.random.rand(4,3)
#W = np.random.randint(5, size=(4,3))

print_vec("重み", W)

# バイアス
b = np.array([0.1, 0.2, 0.3])
print_vec("バイアス", b)

# 入力値
x = np.array([1.0, 5.0, 2.0, -1.0])
print_vec("入力", x)

# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.sigmoid(u)
print_vec("中間層出力", z)
```

```
*** 重み ***
[[0.1 0.2 0.3]
 [0.2 0.3 0.4]
 [0.3 0.4 0.5]
 [0.4 0.5 0.6]]
```

```
*** バイアス ***
[0.1 0.2 0.3]
```

```
*** 入力 ***
[ 1.  5.  2. -1.]
```

```
*** 総入力 ***
[1.4 2.2 3. ]
```

```
*** 中間層出力 ***
[0.80218389 0.90024951 0.95257413]
```

→中間層が3ノードの場合

→中間出力:中間層での計算値をシグモイド関数で活性化し出力

=====

3.順伝播（3層・複数ユニット）

In [7]:

```
# 順伝播 (3層・複数ユニット)

# ウェイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")
    network = {}

    # 試してみよう
    # 各パラメータのshapeを表示
    # ネットワークの初期値ランダム生成

    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.1, 0.4],
        [0.2, 0.5],
        [0.3, 0.6]
    ])
    network['W3'] = np.array([
        [0.1, 0.3],
        [0.2, 0.4]
    ])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2])
    network['b3'] = np.array([1, 2])

    print_vec("重み1", network['W1'])
    print_vec("重み2", network['W2'])
    print_vec("重み3", network['W3'])
    print_vec("バイアス1", network['b1'])
    print_vec("バイアス2", network['b2'])
    print_vec("バイアス3", network['b3'])

    return network

# プロセスを作成
# x: 入力値
def forward(network, x):

    print("##### 順伝播開始 #####")

    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    # 1層の総入力
    u1 = np.dot(x, W1) + b1

    # 1層の総出力
    z1 = functions.relu(u1)

    # 2層の総入力
    u2 = np.dot(z1, W2) + b2

    # 2層の総出力
    z2 = functions.relu(u2)
```

```
# 出力層の総入力
u3 = np.dot(z2, W3) + b3

# 出力層の総出力
y = u3

print_vec("総入力1", u1)
print_vec("中間層出力1", z1)
print_vec("総入力2", u2)
print_vec("出力1", z1)
print("出力合計: " + str(np.sum(z1)))

return y, z1, z2

# 入力値
x = np.array([1., 2.])
print_vec("入力", x)

# ネットワークの初期化
network = init_network()

y, z1, z2 = forward(network, x)
```

\*\*\* 入力 \*\*\*

[1. 2.]

##### ネットワークの初期化 #####

\*\*\* 重み1 \*\*\*

[[0.1 0.3 0.5]  
[0.2 0.4 0.6]]

\*\*\* 重み2 \*\*\*

[[0.1 0.4]  
[0.2 0.5]  
[0.3 0.6]]

\*\*\* 重み3 \*\*\*

[[0.1 0.3]  
[0.2 0.4]]

\*\*\* バイアス1 \*\*\*

[0.1 0.2 0.3]

\*\*\* バイアス2 \*\*\*

[0.1 0.2]

\*\*\* バイアス3 \*\*\*

[1 2]

##### 順伝播開始 #####

\*\*\* 総入力1 \*\*\*

[0.6 1.3 2. ]

\*\*\* 中間層出力1 \*\*\*

[0.6 1.3 2. ]

\*\*\* 総入力2 \*\*\*

[1.02 2.29]

\*\*\* 出力1 \*\*\*

[0.6 1.3 2. ]

出力合計: 3.9

→層を増やしてもやることは同じ

中間層の計算: $u = x * W + b$ とReLU関数で活性化し出力することの繰り返し

=====

4.多クラス分類 (2-3-4ネットワーク)



In [9]:

```
# 多クラス分類
# 2-3-4ネットワーク

# ! 試してみよう_ノードの構成を 3-5-4 に変更してみよう

# ウェイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

    #試してみよう
    #_各パラメータのshapeを表示
    #_ネットワークの初期値ランダム生成

    network = {}
    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.1, 0.4, 0.7, 1.0],
        [0.2, 0.5, 0.8, 1.1],
        [0.3, 0.6, 0.9, 1.2]
    ])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2, 0.3, 0.4])

    print_vec("重み1", network['W1'])
    print_vec("重み2", network['W2'])
    print_vec("バイアス1", network['b1'])
    print_vec("バイアス2", network['b2'])

    return network

# プロセスを作成
# x: 入力値
def forward(network, x):

    print("##### 順伝播開始 #####")
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 1層の総入力
    u1 = np.dot(x, W1) + b1

    # 1層の総出力
    z1 = functions.relu(u1)

    # 2層の総入力
    u2 = np.dot(z1, W2) + b2

    # 出力値
    y = functions.softmax(u2)

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(y)))
```

```
    return y, z1

## 事前データ
# 入力値
x = np.array([1., 2.])

# 目標出力
d = np.array([0, 0, 0, 1])

# ネットワークの初期化
network = init_network()

# 出力
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)

## 表示
print("¥n##### 結果表示 #####")
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("誤差", loss)
```

##### ネットワークの初期化 #####

\*\*\* 重み1 \*\*\*

[[0.1 0.3 0.5]  
[0.2 0.4 0.6]]

\*\*\* 重み2 \*\*\*

[[0.1 0.4 0.7 1. ]  
[0.2 0.5 0.8 1.1]  
[0.3 0.6 0.9 1.2]]

\*\*\* バイアス1 \*\*\*

[0.1 0.2 0.3]

\*\*\* バイアス2 \*\*\*

[0.1 0.2 0.3 0.4]

##### 順伝播開始 #####

\*\*\* 総入力1 \*\*\*

[0.6 1.3 2. ]

\*\*\* 中間層出力1 \*\*\*

[0.6 1.3 2. ]

\*\*\* 総入力2 \*\*\*

[1.02 2.29 3.56 4.83]

\*\*\* 出力1 \*\*\*

[0.01602796 0.05707321 0.20322929 0.72366954]

出力合計: 1.0

##### 結果表示 #####

\*\*\* 出力 \*\*\*

[0.01602796 0.05707321 0.20322929 0.72366954]

\*\*\* 訓練データ \*\*\*

[0 0 0 1]

\*\*\* 誤差 \*\*\*

0.32342029336019423

→中間層の活性化にはReLU関数、出力層にはソフトマックス関数を使用

→最後に確率として出力するために、ソフトマックス関数を用いている

→目標出力と実際の出力を比較すると、誤差はあるものの分類結果は正解している(出力層の中の最大値を採用するとした場合)

=====

5.回帰 (2-3-2ネットワーク)

In [10]:

```

# 回帰
# 2-3-2ネットワーク

# ! 試してみよう_ノードの構成を 3-5-4 に変更してみよう

# ウェイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

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

    print_vec("重み1", network['W1'])
    print_vec("重み2", network['W2'])
    print_vec("バイアス1", network['b1'])
    print_vec("バイアス2", network['b2'])

    return network

# プロセスを作成
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
    z1 = functions.relu(u1)
    # 出力層の総入力
    u2 = np.dot(z1, W2) + b2
    # 出力層の総出力
    y = u2

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(z1)))

    return y, z1

# 入力値
x = np.array([1., 2.])
network = init_network()
y, z1 = forward(network, x)
# 目標出力
d = np.array([2., 4.])

```

```
# 誤差
loss = functions.mean_squared_error(d, y)
## 表示
print("\n##### 結果表示 #####")
print_vec("中間層出力", z1)
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("誤差", loss)
```

##### ネットワークの初期化 #####

\*\*\* 重み1 \*\*\*

```
[[0.1 0.3 0.5]
 [0.2 0.4 0.6]]
```

\*\*\* 重み2 \*\*\*

```
[[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]
```

\*\*\* バイアス1 \*\*\*

```
[0.1 0.2 0.3]
```

\*\*\* バイアス2 \*\*\*

```
[0.1 0.2]
```

##### 順伝播開始 #####

\*\*\* 総入力1 \*\*\*

```
[0.6 1.3 2. ]
```

\*\*\* 中間層出力1 \*\*\*

```
[0.6 1.3 2. ]
```

\*\*\* 総入力2 \*\*\*

```
[1.02 2.29]
```

\*\*\* 出力1 \*\*\*

```
[1.02 2.29]
```

出力合計: 3.9

##### 結果表示 #####

\*\*\* 中間層出力 \*\*\*

```
[0.6 1.3 2. ]
```

\*\*\* 出力 \*\*\*

```
[1.02 2.29]
```

\*\*\* 訓練データ \*\*\*

```
[2. 4.]
```

\*\*\* 誤差 \*\*\*

```
0.9711249999999999
```

→中間層の活性化にはReLU関数、出力層には恒等関数を使用

→回帰の場合は、出力層の活性化関数に恒等関数を使用する

=====

## 6.2値分類（2-3-1ネットワーク）

In [11]:

```
# 2値分類
# 2-3-1ネットワーク

# ! 試してみよう_ノードの構成を 5-10-1 に変更してみよう

# ウェイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

    network = {}
    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.2],
        [0.4],
        [0.6]
    ])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1])
    return network

# プロセスを作成
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
    z1 = functions.relu(u1)
    # 出力層の総入力
    u2 = np.dot(z1, W2) + b2
    # 出力層の総出力
    y = functions.sigmoid(u2)

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(z1)))

    return y, z1

# 入力値
x = np.array([1., 2.])
# 目標出力
d = np.array([1])
network = init_network()
y, z1 = forward(network, x)
# 誤差
loss = functions.cross_entropy_error(d, y)
```

```
## 表示
print("\n##### 結果表示 #####")
print_vec("中間層出力", z1)
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("誤差", loss)
```

##### ネットワークの初期化 #####

##### 順伝播開始 #####

\*\*\* 総入力1 \*\*\*

[0.6 1.3 2. ]

\*\*\* 中間層出力1 \*\*\*

[0.6 1.3 2. ]

\*\*\* 総入力2 \*\*\*

[1.94]

\*\*\* 出力1 \*\*\*

[0.87435214]

出力合計: 3.9

##### 結果表示 #####

\*\*\* 中間層出力 \*\*\*

[0.6 1.3 2. ]

\*\*\* 出力 \*\*\*

[0.87435214]

\*\*\* 訓練データ \*\*\*

[1]

\*\*\* 誤差 \*\*\*

0.13427195993720972

→中間層の活性化にはReLU関数、出力層にはシグモイド関数を使用

→2値分類の場合は、シグモイド関数を使用する

In [ ]: