3章 ニューラルネットワーク

NAL研
具志堅凌河, 大城紳之助, 松本一馬
津波大輝, 島袋寛都

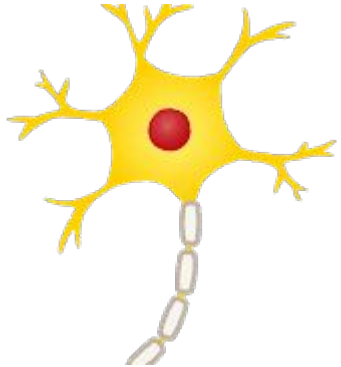
学習の目的

ニューラルネットワークについての知識を深める

3.1 パーセプトロンからニューラルネットワークへ

- 概要
  - ニューラルネットワークは前章で扱った単純パーセプトロンと共通する点が多い。
  - 簡単なニューラルネットワークと単純パーセプトロンを例示し、互いの異なる点を大まかに見る

- 目次
  - ニューラルネットワークの例示
  - 単純パーセプトロンの復習
  - 活性化関数の登場



3.1.1 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例を右に示す。

・左から入力層、中間層(隠れ層)、出力層の3つから構成されているのが、例示した物である。

・図からわかるように、2章で扱ったパーセプトロンと酷似している。

・ニューロンのつながり方に関しては、パーセプトロンともなんら変わりない構造である。

ニューラルネットワークではどのように
信号を伝達するのかが重要になってくる。

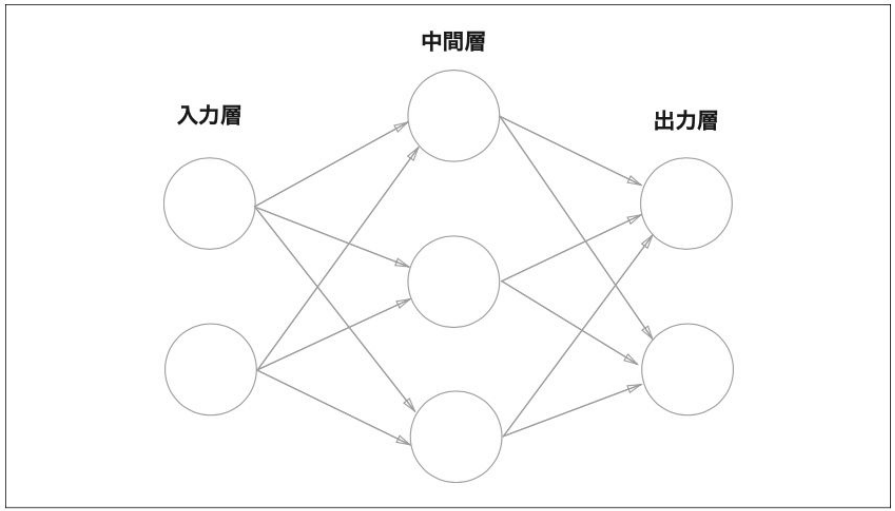
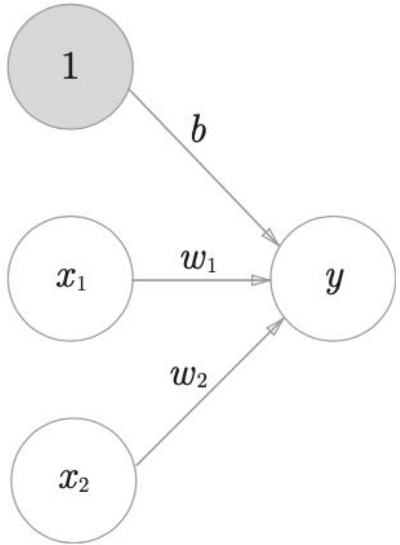


図3-1　ニューラルネットワークの例

3.1.2 パーセプトロンの復習

右図のような単純パーセプトロンを考える。



これは入力にx1,x2とバイアスを受け取り、yを出力する。
パーセプトロンの出力は0,1の2値であるため、式で表すと

$$y = \begin{cases} 0 & (b + w_1x_1 + w_2x_2 \leqq 0) \\ 1 & (b + w_1x_1 + w_2x_2 > 0) \end{cases} \tag{3.1}$$

となる。

またbはバイアス、w1,w2は各信号の重みを表す記号である。

上記の式を簡略化する。0を超えたら1を出力し、そうでなければ0を出力する動作を関数hとして表すと、

式を以下のように簡略化し、書き換えることができる。

$$y = h(b + w_1x_1 + w_2x_2) \tag{3.2}$$

$$h(x) = \begin{cases} 0 & (x \leqq 0) \\ 1 & (x > 0) \end{cases} \tag{3.3}$$

3.1.3 活性化関数の登場

3.1.2で示したh(x)のような、入力信号の総和を出力信号に変換する関数のことを、一般に**活性化関数**と呼ぶ。

「活性化」という名前の意味通り、活性化関数は入力信号の総和が、どのように活性化(発火)するかということを決定する役割を持つ。

上で示した式3.2を書き換え、活性化関数での役割を分割し、別の式として表すと

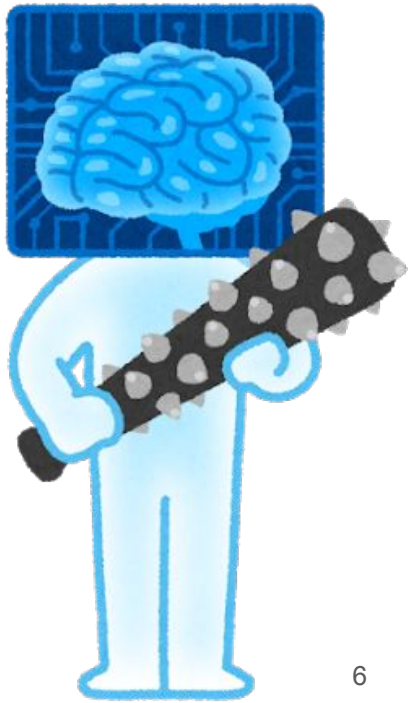
重み付きの入力信号の総和を計算する式として

$$a = b + w_1 x_1 + w_2 x_2 \tag{3.4}$$

和が活性化関数により変換する式として

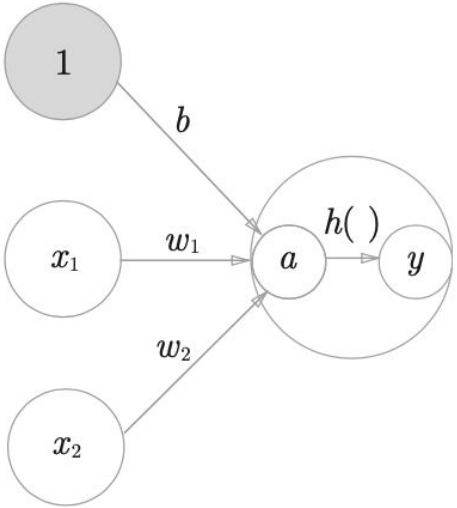
$$y = h(a) \tag{3.5}$$

と表せる。



3.1.3 活性化関数の登場

定義した3.4,3.5の式を踏まえ、活性化関数のプロセスを明示的に表すと
右のようになる。



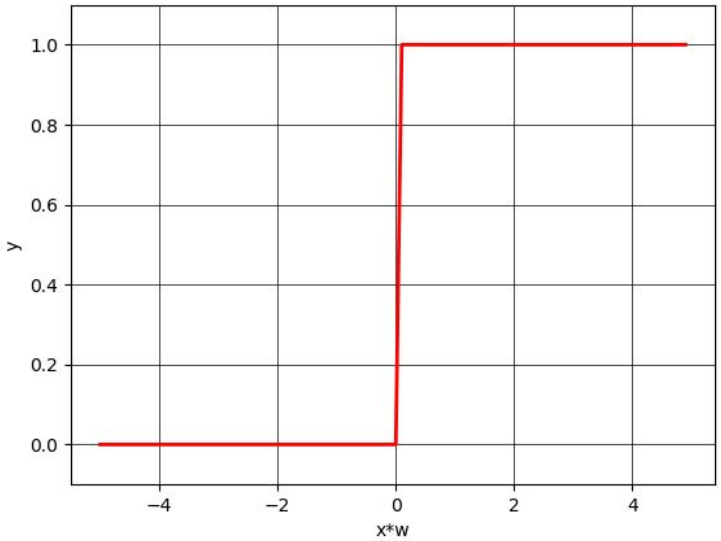
これは重み付き信号の和の結果がaというノード(ニューロン)になり、
活性化関数h()によってyというノードに変換されていることを示している。

次節では、活性化関数について解説する。この活性化関数について
知ることが、パーセプトロンとニューラルネットワークの違いを
理解するのに必要となるであろう。

3.2 活性化関数(概要, 活性化関数の紹介, 活性化関数の実装方法, 非線形関数)

- 概要
  - 活性化関数とは, 閾値を境にして出力が切り替わる関数である。いろんな種類があり, 活性化関数の違いによって, 同じ入力でも出力が異なることになります。
  - 本節では, ステップ関数, シグモイド関数, ReLU関数の3種類を紹介している
  - 非線形関数についての知識もニューラルネットワークでは , 必要になっています。

- 活性化関数の紹介(ステップ, シグモイド, ReLU)



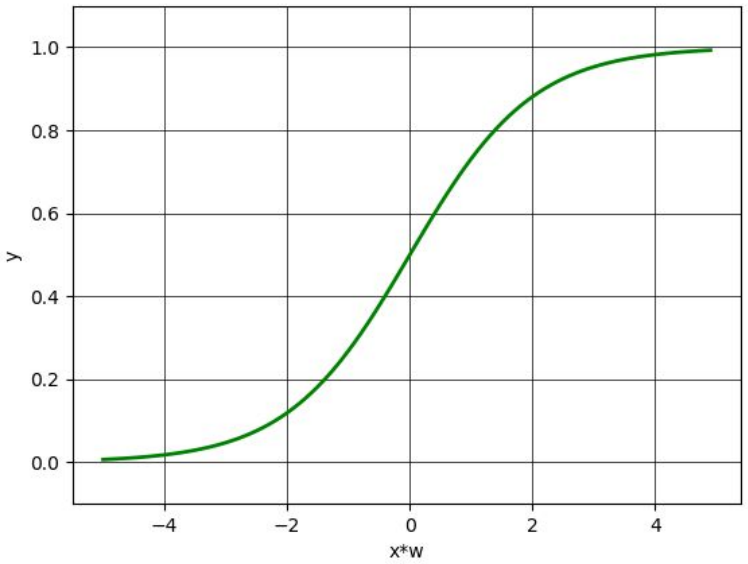
- ステップ関数・・・パーセプトロンに使用されている活性化関数のひとつ。　入力した値が0以下であれば, 0を出力する。0より大きければ, 1を出力する関数

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

- ここでの xは入力ですが, 重み付きの入力同士を足し合わせたもの。この後に, 説明する活性化関数も同様。

3.2 活性化関数(概要, 活性化関数の紹介, 活性化関数の実装方法, 非線形関数)

- 活性化関数の紹介(ステップ, シグモイド, ReLU)



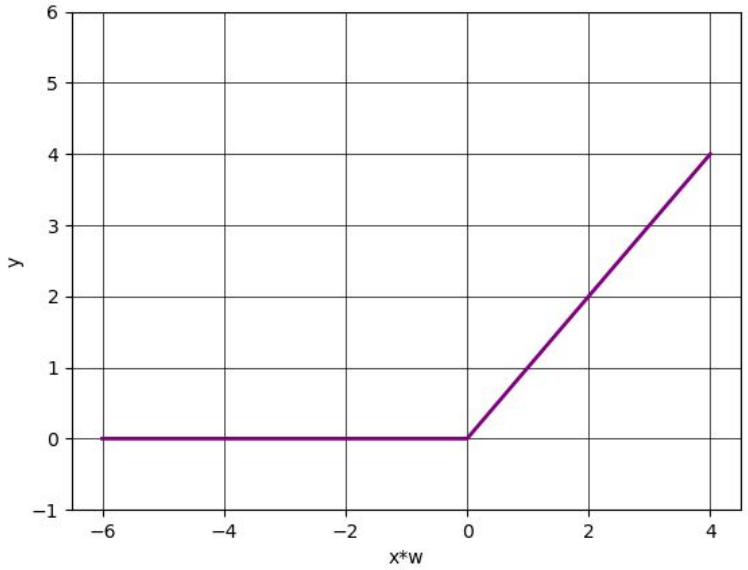
- シグモイド関数・・・ニューラルネットワークでよく用いられる活性化関数の1つ。

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

- ステップ関数と比較すると, グラフの形が滑らかな曲線であり, 0と1の間であれば, 0と1以外の数値でも出力可能 (ex 0.787, 0.788, 0.001, 0, 1 etc...)

3.2 活性化関数(概要, 活性化関数の紹介, 活性化関数の実装方法, 非線形関数)

- 活性化関数の紹介(ステップ, シグモイド, ReLU)



- ReLU関数・・・最近熱い活性化関数。

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

- “***Rectified Linear Unit***”の略。入力が0を超えていれば, 入力をそのまま出力, 0以下であれば, 0を出力する関数。

- 活性化関数の実装方法 (ステップ, シグモイド, ReLU)
- ステップ関数・・・入力した値が 0以下であれば, 0を出力する。0より大きければ, 1を出力する関数

```
def step_function(x):
    if x>0:
        return 1
    else:
        return 0
```

これでもいいけど, 引数に配列を取ることができないから, 計算めんど…

```
def step_function(x):
    y = x > 0
    return y.astype(np.int)
```

3.2 活性化関数(概要, 活性化関数の紹介, 活性化関数の実装方法, 非線形関数)

- 活性化関数の実装方法 (ステップ, シグモイド, ReLU関数)
- シグモイド関数・・・出力が 1/1+exp(-x)を計算すると得られる関数

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

- exp(-x)は指数eを-x乗した数, e^(-x)である。numpyに用意されているnp.exp()を使うと簡単に実装できる。また, np.exp(x)の引数にnumpy配列を入力すると, 結果を配列で返してくれる。
- numpy配列を入力とすると, 配列の要素ごとに計算 (ブロードキャスト)し, 結果を配列として出力してくれる。
- ブロードキャストとは, スカラ値とnumpy配列で計算が行われると, スカラ値と配列の各要素どうしで計算が行われる。

- 活性化関数の実装方法 (ステップ, シグモイド, ReLU)
- ReLU関数・・・入力が0を超えていれば, 入力をそのまま出力, 0以下であれば, 0を出力する関数。

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

- np.maximum()は引数にとった値の中から最大値を出力する。np.exp()と同様に, 引数に配列を入力すると, 配列の要素ごとに演算し, 配列を生成して返してくれる。
- 例えば, x = np.array([2.3, -0.4, 7.2])を入力する。最初に, np.maximum()の処理として0と2.3のどちらが最大値かを選択, その次に, 0と-0.4のどちらかが最大値かを選択, 最後の要素についても同様に処理を行い, 結果を配列として **[2.3, 0, 7.2]**として返してくれる。

3.2 活性化関数(概要, 活性化関数の紹介, 活性化関数の実装方法, 非線形関数)

- 非線形関数

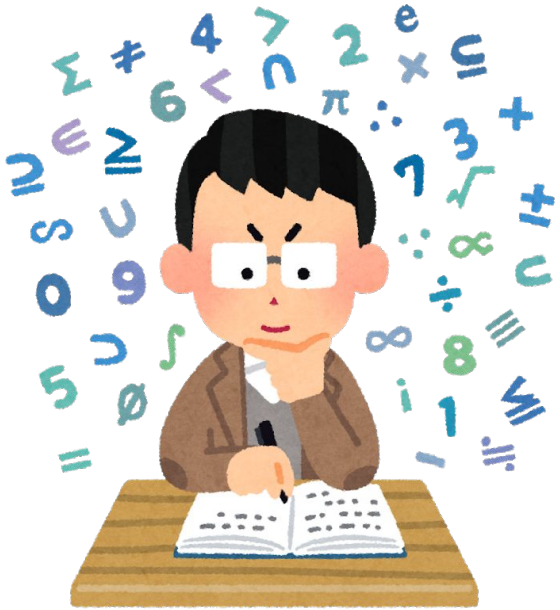
- 簡単に言えば, グラフの形が1つの直線であらわせないような関数。例えば, 紹介したステップ関数, シグモイド関数, ReLU関数などは非線形関数であると言えます。また, 2次関数y(x) = ax^2 + bx + cなども非線形関数です。

- ニューラルネットワークでは, 活性化関数を非線形関数にする必要があります。線形関数を活性化関数にしようしてしまうと, 隠れ層がないネットワークで実現できてしまうため, ニューラルネットワークで層を深くする必要がなくなってしまう…。

3.3 多次元配列の計算

- 概要
  - NumPyの多次元配列を利用することで、ニューラルネットワークの実装を効率的に進めることができる
  - ここでは多次元配列や行列の内積について説明する

- 目次
  - 多次元配列
  - 行列の内積
  - ニューラルネットワークの内積



3.3.1　多次元配列

多次元配列・・・数字がN次元上に並んだもの。数学でいうベクトル(1次元)や行列(2次元)を表現できる。

例：NumPyで行列(二次元配列)を作成

```
>>> import numpy as np
>>> B = np.array([[1,2], [3,4], [5,6]])
>>> print(B)
[[1 2]
 [3 4]
 [5 6]]
>>> np.ndim(B)
2
>>> B.shape
(3, 2)
```

NumPyの関数説明

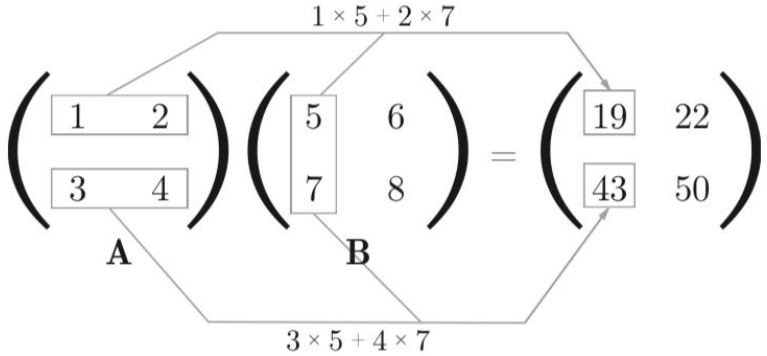
・np.array(変数)　→　引数に配列を指定することでN次元配列を作成することができる。

・np.ndim(変数)　→　引数に指定した多次元配列の次元数を取得できる。

・変数.shape　→　多次元配列の形状をタプルで取得できる。
ここでは行数3、列数2の行列が作成されている。

・np.dot(変数1, 変数2)　→　引数で指定した二つの行列やベクトルの内積を求める。
変数1.dot(変数2)でも可。引数の順番で計算結果が変わる。

3.3.2 行列の内積



行列の内積の特徴

・左の行列の行と右の行列の列を一つずつ選択し、要素ごとの積とその和を求める

・左のN行目と右のM列目を選択した場合、内積結果の行列のN行M列目の要素となる

・左の行列の列数と右の行列の行数が一致しなければ内積を計算できない

・左の行数と右の列数が内積結果の行数と列数となる

・二つの行列の計算順序を逆にすれば計算結果も変わる

・行列とベクトルでも内積は求められる

例：上図の行列の内積をNumPyで実装

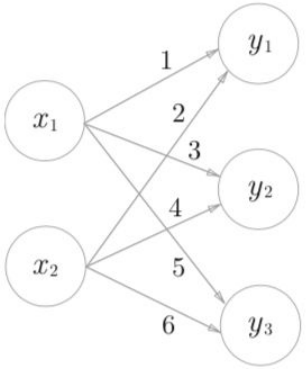
```
>>> A = np.array([[1,2], [3,4]])
>>> A.shape
(2, 2)
>>> B = np.array([[5,6], [7,8]])
>>> B.shape
(2, 2)
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

3.3.3 ニューラルネットワークの内積

ニューラルネットワークの計算でNumPyを用いた際の利点

・内積をfor文などを利用せずに簡単に実装可能

・多次元配列を利用できるため、入力や出力が多くても一度に計算できる



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

$$\mathbf{X} \quad \mathbf{W} \quad = \quad \mathbf{Y}$$

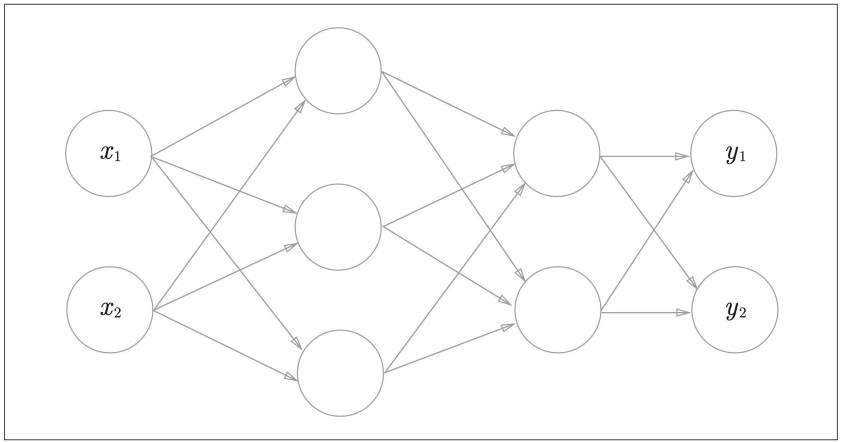
$$2 \quad 2 \times 3 \quad 3$$

一致

例：上図のニューラルネットワークの内積をNumPyで実装(バイアスや活性化関数は省略)

```
>>> X = np.array([1, 2])
>>> W = np.array([[1, 3, 5], [2, 4, 6]])
>>> Y = np.dot(X, W)
>>> print(Y)
[ 5 11 17]
```

3.4 3層のミューラルネットワークの実装

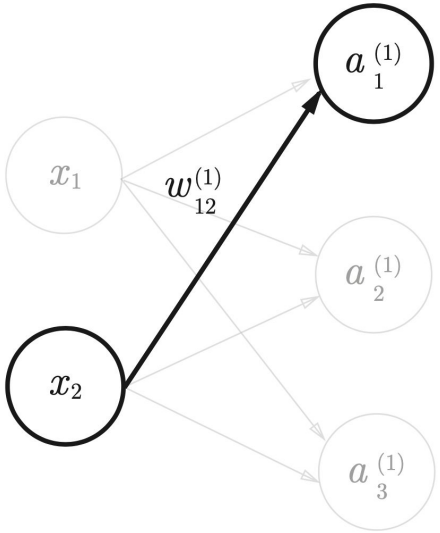


- 目次
  - 使用する記号の確認
  - 各層の信号伝達の実装
  - 実装まとめ

- 概要
  - 3節では、これまでに説明したNumPyの多次元配列計算を用いて実際に3層ニューラルネットワークを実装していく。
    実装に際し、具体的な計算や活性化関数の使い方を詳しく説明していく。

3.4.1 3層のミューラルネットワークの実装（使用する記号の確認）

本節では、ニューラルネットワークの
処理を説明するために、右図のような
記号を使用する。
これらの記号は本節だけで使用するもの
であり、特に覚える必要はない。



$w_{12}^{(1)}$

- 第1層目の重み
- 前層の2番目のニューロン
- 次層の1番目のニューロン

3.4.2 3層のミューラルネットワークの実装（各層の信号伝達の実装）

まず、第1層目の1番目のニューロンに注目して説明していく。
a_1は以下のような式で表せる

$$a^{(1)}_1 = w^{(1)}_{11} x_1 + w^{(1)}_{12} x_2 + b^{(1)}_1$$

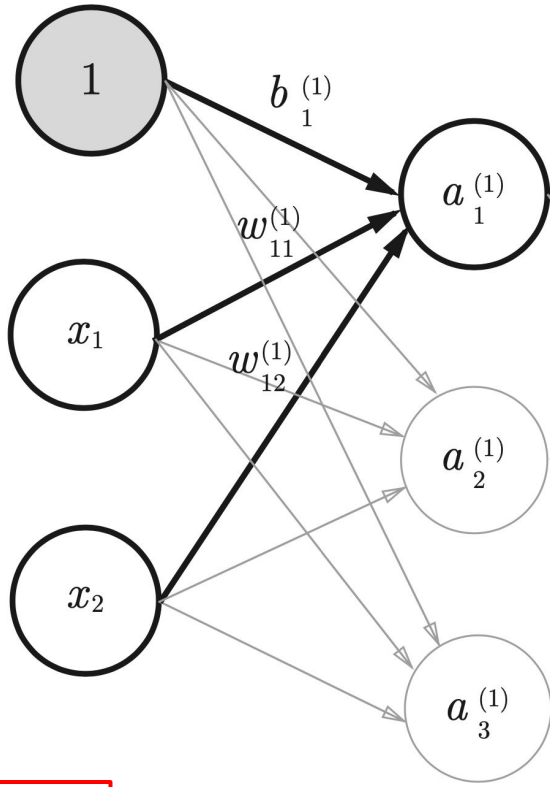
さらに、行列の内積によって表すと以下の式になる

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

ここで、$A^{(1)}$, X, $B^{(1)}$, $W^{(1)}$ は以下のようになっている

$$\mathbf{A}^{(1)} = \begin{pmatrix} a_1^{(1)} & a_2^{(1)} & a_3^{(1)} \end{pmatrix}, \mathbf{X} = \begin{pmatrix} x_1 & x_2 \end{pmatrix}, \mathbf{B}^{(1)} = \begin{pmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{pmatrix}$$

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

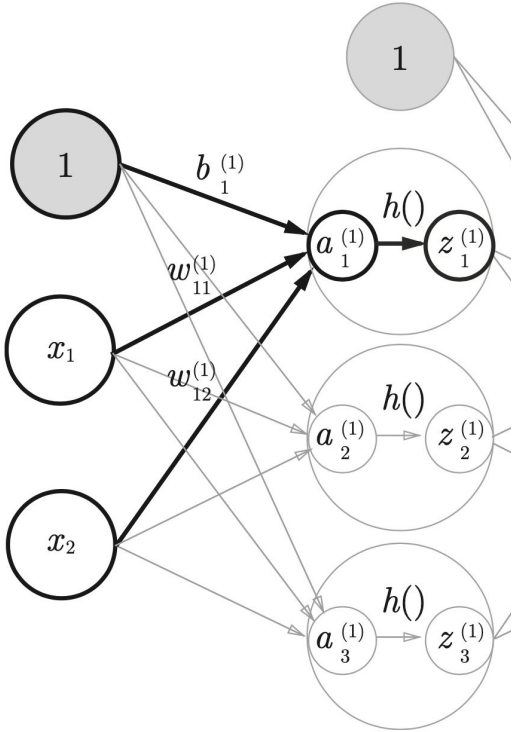


(1,3) = (1,2) × (2,3) + (1,3)

3.4.2 3層のミューラルネットワークの実装（各層の信号伝達の実装）

次に、第1層の活性化関数の処理について説明していく。
この活性化関数の処理は右図のようになる。

ここでは、隠れ層での重み付き和を a、活性化関数をh()で表し、
活性化関数で変換された信号を zで表している。
また、活性化関数にシグモイド関数を使用する。

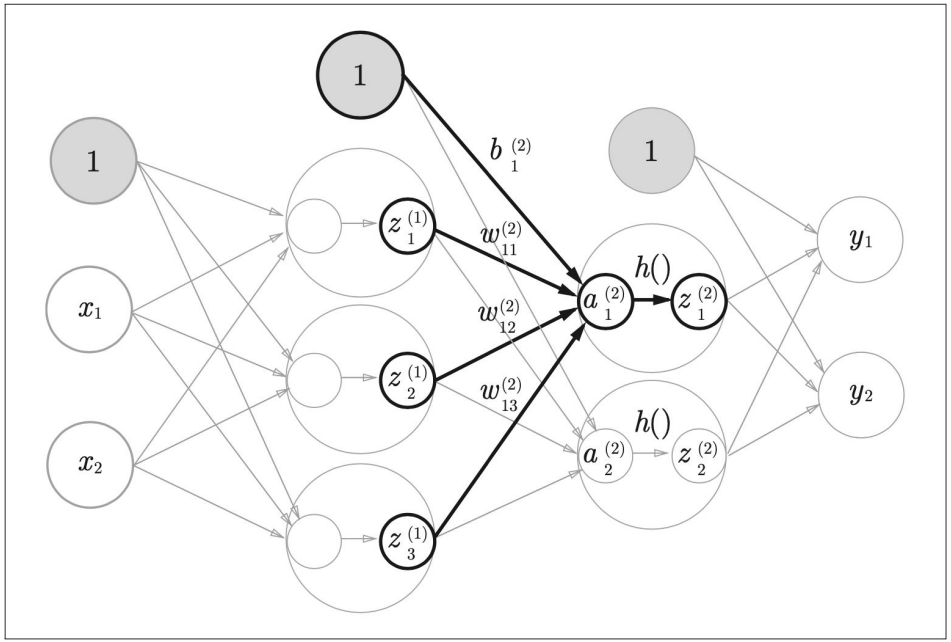


これまでの処理をPythonで実装すると以下のようになる。

```
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])

print(W1.shape) # (2, 3)
print(X.shape)  # (2,)
print(B1.shape) # (3,)

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

print(A1) # [0.3, 0.7, 1.1]
print(Z1) # [0.57444252, 0.66818777, 0.75026011]
```

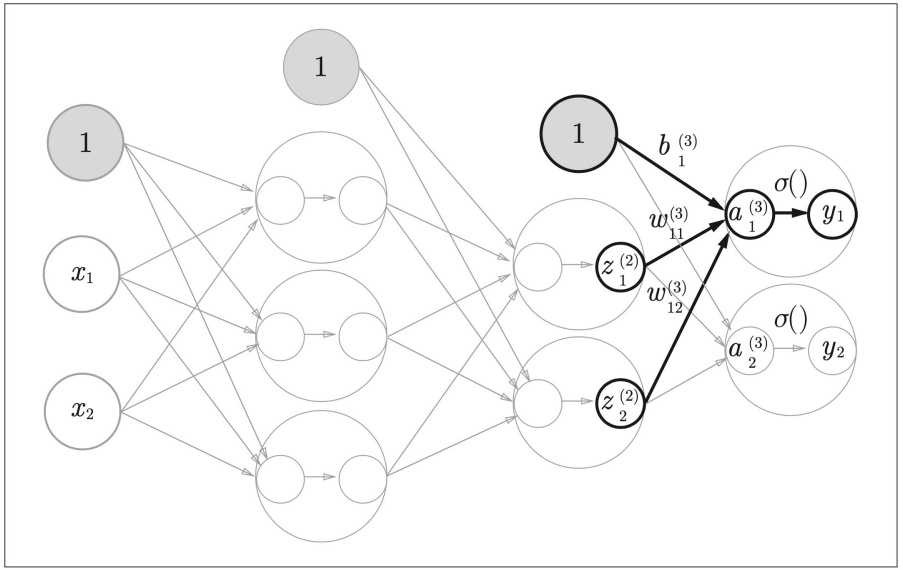
3.4.2 3層のミューラルネットワークの実装（各層の信号伝達の実装）

第1層から第2層までの処理については先程の処理と殆ど変わらないため、省略する。
異なる部分としては、入力がXから第1層の出力Z1に変わっていることである。
一応、処理は右図のようになっている。



3.4.2 3層のミューラルネットワークの実装（各層の信号伝達の実装）

次に、第2層から出力層への信号伝達を見ていく。
この処理も今までと大きくは変わらないが、最後に使用する活性化関数がこれまでとは異なる。
この伝達処理は右図のようになっている。



今回、最後に使用する活性化関数は恒等関数と呼ばれるものを使用する。これは入力をそのまま出力する関数であり、他の関数と区別するためにh()ではなくσ()として表す。

この恒等関数をPythonで実装すると以下のようになる

```
def identity_function(x):
    return x
```

3.4.2 3層のミューラルネットワークの実装（実装のまとめ）

全ての処理の説明が終わったので今までの実装を
まとめて書いてみる。
右図が実際にまとめたものを書いたコードである。

Init_network()関数で重みとバイアスを初期化

forward()関数で入力から出力までの処理をまとめて
行う

```python
import numpy as np

def identity_function(x):
    return x

def sigmoid(x):
    return 1/(1+np.exp(-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) # [ 0.31682708 0.69627909]
```

3.5 出力層の設計

この3.5節ではソフトマックス関数について説明します。

ニューラルネットワークは回帰、分類問題の両方に対して用いることができる。
問題により出力層の活性化関数を変更する必要がある。

- 回帰問題
  - 恒等関数
- 分類問題
  - ソフトマックス関数

そもそも回帰問題、分類問題とは？？

回帰問題

入力データ(連続的な)がどの様な値になるかという問題です。

入力データ(連続値)

0, 1, 1, 2, 3, 5, 8, 13,
21, 34, 55, 89

出力データ(連続値)

0, 1, 1, 2, 3, 5, 8, 13,
21, 34, 55, 89,144

分類問題

予測したい入力データがどのクラスに属するかという問題です。

クラス

入力データ(画像)

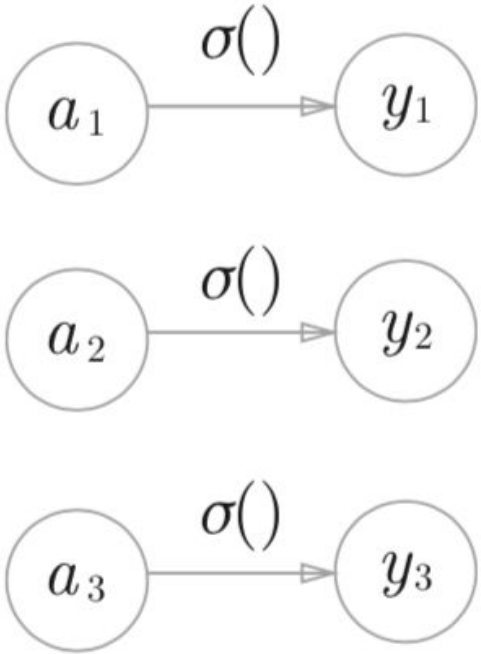
女性

男性

回帰問題で使用される恒等関数とは

恒等関数は、入力をそのまま出力する関数

ニューラルネットワークの図で表すと



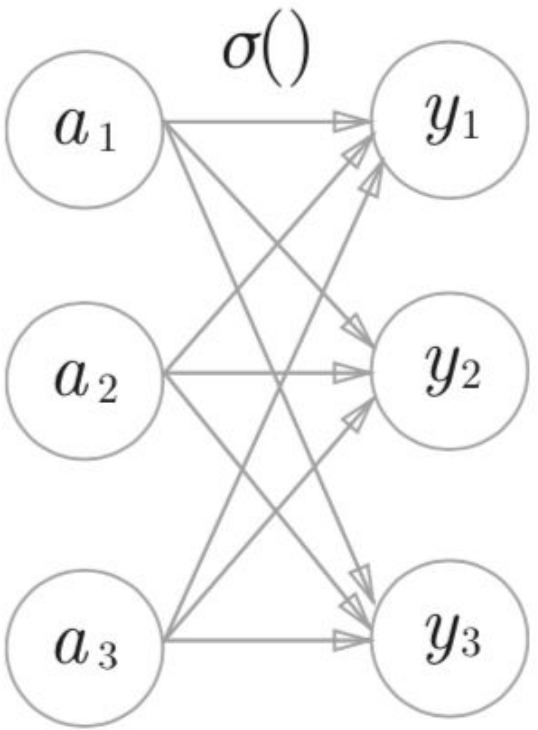
分類問題で使用されるソフトマックス関数とは
以下の式で表される

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

exp(e)はe^x表す指数関数です。
出力 層が全部でn 個あるとして、
k 番目の出力 yk を求める計算式を表しています。

ニューラルネットワークの図で表すと
右図の様になる

すべての入力信号から矢印による結びつきがあり、
式の分母から分かるように、出力の各ニューロンが、
すべての入力 信号から影響を受けることになるからです。



実際にpythonでソフトマックス関数を実装すると

```
1.  def softmax(a):
2.      exp_a = np.exp(a)
3.      sum_exp_a = np.sum(exp_a)
4.      y = exp_a / sum_exp_a
5.
6.      return y
```

1行目、入力信号を引数としてもらう
2行目、入力信号の指数計算　(e^a)
3行目、指数計算後の値の総和　(式の分母の部分)
4行目、入力信号それぞれに対し式を適用

```
>>> a = np.array([0.3, 2.9, 4.0])
>>>
>>> exp_a = np.exp(a) # 指数関数
>>> print(exp_a)
[  1.34985881  18.17414537  54.59815003]
>>>
>>> sum_exp_a = np.sum(exp_a) # 指数関数の和
>>> print(sum_exp_a)
74.1221542102
>>>
>>> y = exp_a / sum_exp_a
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
```

ソフトマックス関数には実装上問題点がある

このままだとe^1000の計算などをする必要になった場合、
計算結果が大きすぎてコンピューターでは表示できず「 inf」が結果として返ってくる。

これをオーバーフローという

この問題を改善した式が

$$
\begin{aligned}
y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} &= \frac{C \exp(a_k)}{C \sum_{i=1}^{n} \exp(a_i)} \\
&= \frac{\exp(a_k + \log C)}{\sum_{i=1}^{n} \exp(a_i + \log C)} \\
&= \frac{\exp(a_k + C')}{\sum_{i=1}^{n} \exp(a_i + C')}
\end{aligned}
\tag{3.11}
$$

$$
\begin{aligned}
y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)} &= \frac{C\exp(a_k)}{C\sum_{i=1}^{n}\exp(a_i)} \\
&= \frac{\exp(a_k + \log C)}{\sum_{i=1}^{n}\exp(a_i + \log C)} \\
&= \frac{\exp(a_k + C')}{\sum_{i=1}^{n}\exp(a_i + C')}
\end{aligned}
\tag{3.11}
$$

式の最初の変形で分母、分子に定数 Cを掛けている。
次の変形で定数Cを指数関数exp()の中に移動
logCをC'に置き換えている。

この式が示しているのは、ソフトマックス関数に何らかの定数を足し算、引き算しても結果が変わることは無いということです。

C'にはオーバーフロー対策として入力信号の中で最大値を用いることが一般的です。

$$a^x = p$$

$$x = \log_a p$$

上式の対数の定義を用いて
以下の式が成り立つ

$$\log_e C = \log_e C$$

$$e^{\log_e c} = C$$

オーバーフロー対策を適用した例として

```
>>> a = np.array([1010, 1000, 990])
>>> np.exp(a) / np.sum(np.exp(a)) # ソフトマックス関数の計算
array([ nan,  nan,  nan])         # 正しく計算されない
>>>
>>> c = np.max(a) # 1010
>>> a - c
array([  0, -10, -20])
>>>
>>> np.exp(a - c) / np.sum(np.exp(a - c))
array([  9.99954600e-01,   4.53978686e-05,   2.06106005e-09])
```

この例で示すように普通に計算した場合
nan(not a number)になるところを入力信号
の最大値(定数C)を分母、分子から引くことで
対策ができる。

対策を踏まえて関数を実装すると

```
def softmax(a):
    c = np.max(a)
    exp_a = np.exp(a - c) # オーバーフロー対策
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a

    return y
```

これまではソフトマックス関数の実装方法を見てきました。

ではソフトマックス関数の特徴は何でしょう？

```
>>> a = np.array([0.3, 2.9, 4.0])
>>> y = softmax(a)
>>> print(y)
[ 0.01821127  0.24519181  0.73659691]
>>> np.sum(y)
1.0
```

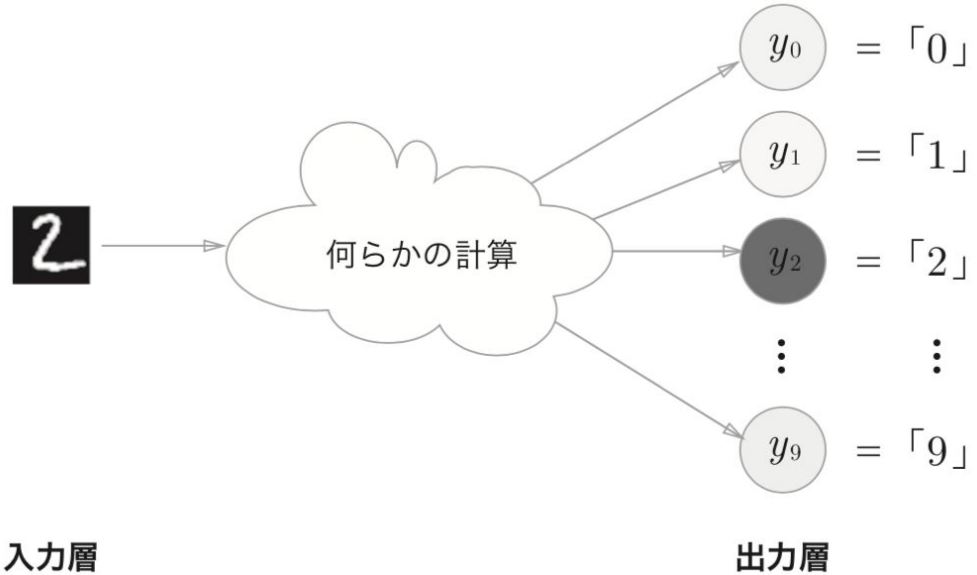
上記の例のようにソフトマックス関数は **0から1.0の間の実数**になり、
また出力の総和は**1**になります。

ここが特徴的な部分で出力を確率として解釈することができます。

しかし、注意点として入力信号の大小関係はソフトマックス関数を通
しても変わることは無いということです。

分類問題の出力層で決めなければいけないものとしてニューロンの数があります。

分類問題ではクラスの個数に応じてニューロンの数を決めます。



上の図では10クラス分類問題なので出力層のニューロンの個数を 10個に設定します。

3.6 手書き数字認識

- 人が書いた数字を画像として入力し，その画像の数字を予測する。画像は，MNISTデータセットという手書き数字の画像セットを使用します。

- 本節では，学習済みのパラメータを用いて，ニューラルネットワークの「推論処理」を実装している。

3.6 手書き数字認識

- MNISTデータセット・・・MNISTデータセットとは，0から9までの数字画像から構成されている。訓練画像が60,000枚，テスト画像が10,000枚用意されており，それらの画像を使って，学習を行い，学習したモデルでテスト画像に対して正しく分類できるかを計測する。

- MNISTデータセットの画像データはサイズが 28×28のグレー画像(1チャンネル)で，各ピクセルは，0から255までの値をとる。それぞれの画像にはどの数字が書いているかを対応させるラベルがついている。

3.6 手書き数字認識

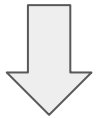
- 手書き数字認識の手順

1. MNISTデータセットの読み込み
2. 推論処理を行うニューラルネットワークの実装
3. バッチ処理

3.6 手書き数字認識

- MNISTデータセットの読み込み

- 当書が提供するソースコードにある load_mnist.pyに定義されているload_mnist関数をインポートします。その後 , load_mnist関数によって , MNISTデータセットを読み込む。



```
import sys, os
sys.path.append(os.pardir)
from dataset import load_mnist

(X_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize = False)
```

3.6 手書き数字認識

- load_mnist関数は「(訓練画像, 訓練ラベル), (テスト画像, テストラベル)」の形で, 読み込んだMNISTデータをnumpy配列として返します。
- 引数flattenは入力画像のピクセルを 0.0~1.0の値に正規化するかどうかを設定できる。
- 引数normalizeは入力画像を1次元配列にするかどうかを設定できる
- ここにはないが, one-hot-labelという引数もある。ラベルを one-hotで表現するかどうかを設定できる

```
import sys, os
sys.path.append(os.pardir)
from dataset import load_mnist

(X_train, t_train), (x_test, t_test) = load_mnist(flatten=True, normalize = False) ←この部分
```

3.6 手書き数字認識

- PILモジュールを使用した , MNISTデータの確認 (ソースコードは長いので割愛)当書のソースコード ch03/mnist_show.pyを実行するとMNISTデータセットの画像が出力される。

- mnist_show.pyで定義されているshow_img関数は, MNISTデータセットを読み込み, その画像データを入力し , 出力させている。画像データは , numpy配列であり, Image.fromarray関数によってPIL用のオブジェクトに変換している。

3.6.2 ニューラルネットワークの推論処理

MNISTデータセットに対して、推論処理を行うニューラルネットワークを実装する。

コードはここのch03のneuralnet_mnist.py

ネットワークは入力層が画像サイズである28 * 28 の784、出力層が0~9の10個のニューロンから構成する。

隠れ層は2つあり、一つ目が50個、二つ目が100個のニューロンを持つ。またこのニューロンの個数は任意に設定した物である。

コードの説明は省略する。(これぐらい解かって)

3.6.2 ニューラルネットワークの推論処理

本実装では入力画像に対し各ピクセル値を0~1の範囲に変換している。(正規化)

このような正規化などの、入力データに対し、何らかの決まった処理を行う手法を前処理という。

- 前処理の例
  - 正規化
  - 標準化
  - 主成分分析(PCA)を用いた次元圧縮
  - etc

あげればまだ種類はあるが、与えられた問題に対し、適した前処理をする必要がある。

脳死で前処理をして良いのはNaNなどの欠損値を取り除く時など限られている。たぶん。

3.6.3 バッチ処理

- バッチ処理とは

ニューラルネットワークで学習する際には、データを入力する必要がある。

その際、毎回データを1つずつ入力し、出力を待つよりも効率の良い手法として、バッチ処理が用いられる。

バッチ処理とは、複数のデータをひとまとめにし、入力として与える手法のことである。

数値計算を扱うライブラリの多くは、大きな配列の計算を効率よく処理できるよう、高度な最適化がされている。そのため、バッチ処理をし、複数のデータをまとめ、一度に計算した方が効率が良い。

(今回の教科書のコード的にはミニバッチのような気がする)

実際に今回のMNISTにバッチ処理を実装したコードの説明などは、省略する。

3.7 まとめ

- ニューラルネットワークは、前章のパーセプトロンと、ニューロンの信号が階層的に伝わるという点で同じ
- 次のニューロンへ信号を送信する際に、信号を変化させる活性化関数に大きな違いが現れる。
- ニューラルネットワークでは活性化関数が滑らかに変化するシグモイド関数、パーセプトロンでは信号が急に変化するステップ関数を使用。この違いが学習に影響を及ぼす。これについては4章で。
- 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使用、出力層のニューロンの個数を分類するクラス数に設定する。