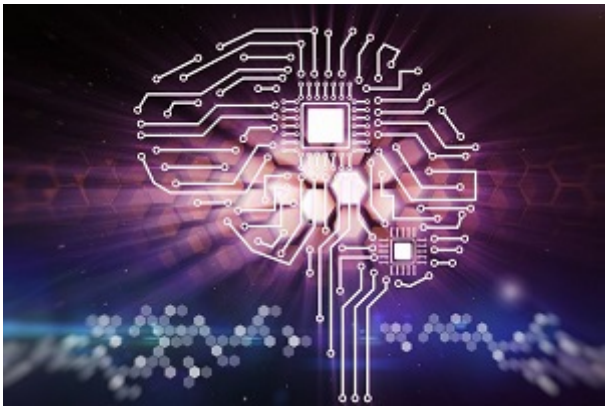
第12章　多層人工ニューラルネットワークをーから実装

［第2版］Python機械学習プログラミング 達人データサイエンティストによる理論と実践
(https://book.impress.co.jp/books/1117101099)



12.1　人口ニューラルネットワークによる複雑な関数のモデル化

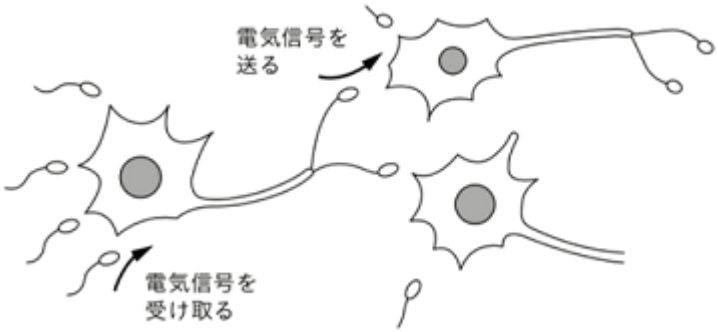
■ ディープラーニングとは？

生物の神経細胞を模したアルゴリズム「ニューラルネットワーク」が主流の技術です。
いま最も高い精度が出やすいと注目されています。

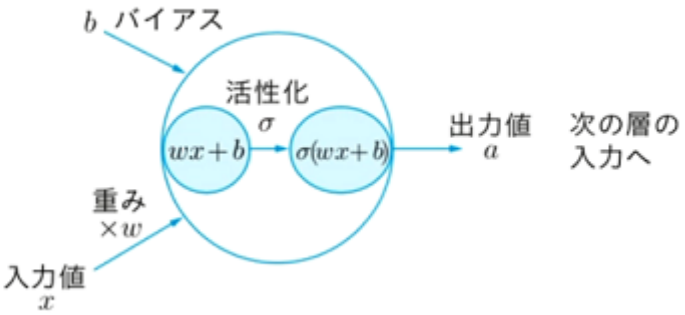
■ ニューロン

ニューロンとは？

生物の神経細胞
ニューロンの仕組み



神経細胞を模した
人工ニューロンの仕組み



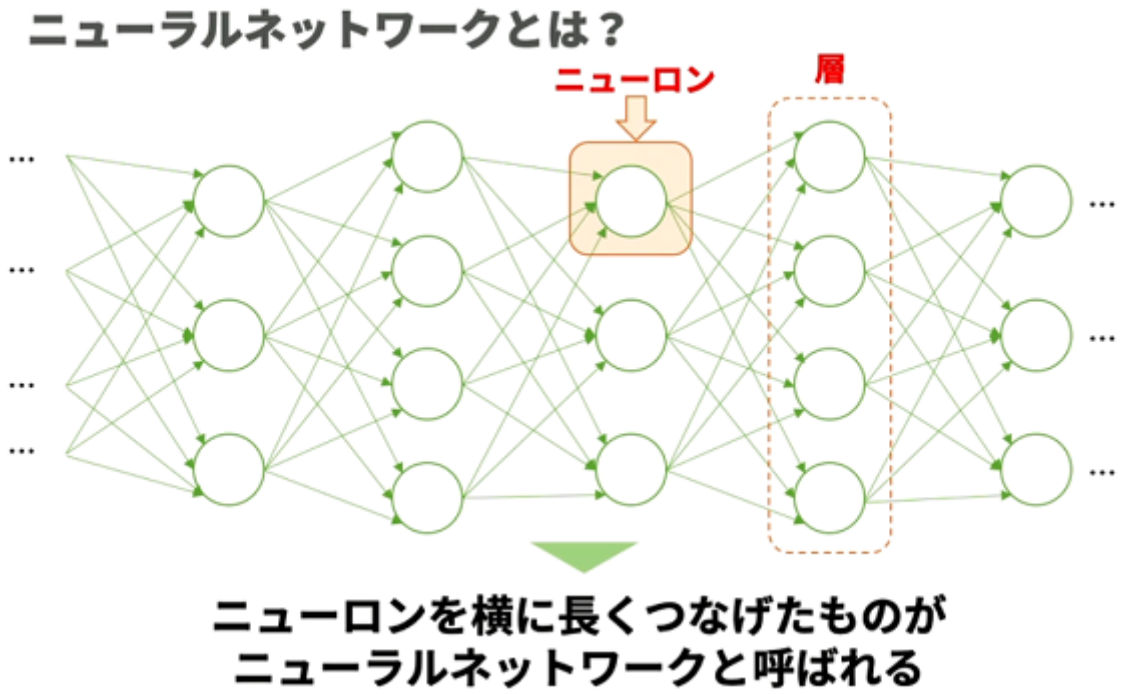
神経細胞は複数の神経細胞から電気信号を受けます。
その電気信号の和が、ある値を超えると発火して、次の神経細胞に電気信号を送ります。

この神経細胞を人工的に再現したものが、ノード（ニューロン／ユニット）です。
電気信号の代わりに数値情報を受け取り、発火（活性化）して、次のノードに情報を送ります。

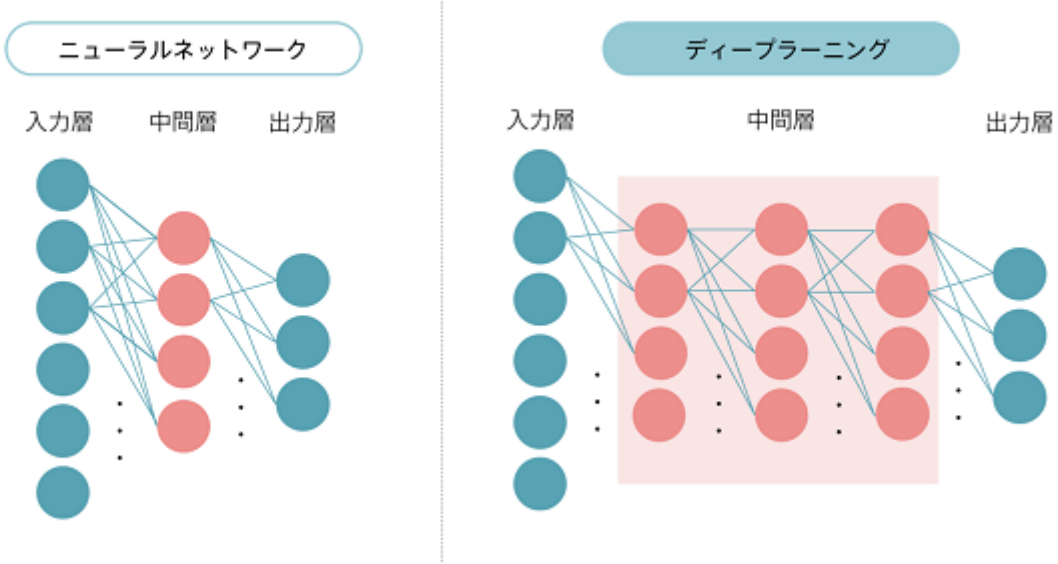
活性化については、後ほど説明します。

■ ニューラルネットワーク（NN）とは？

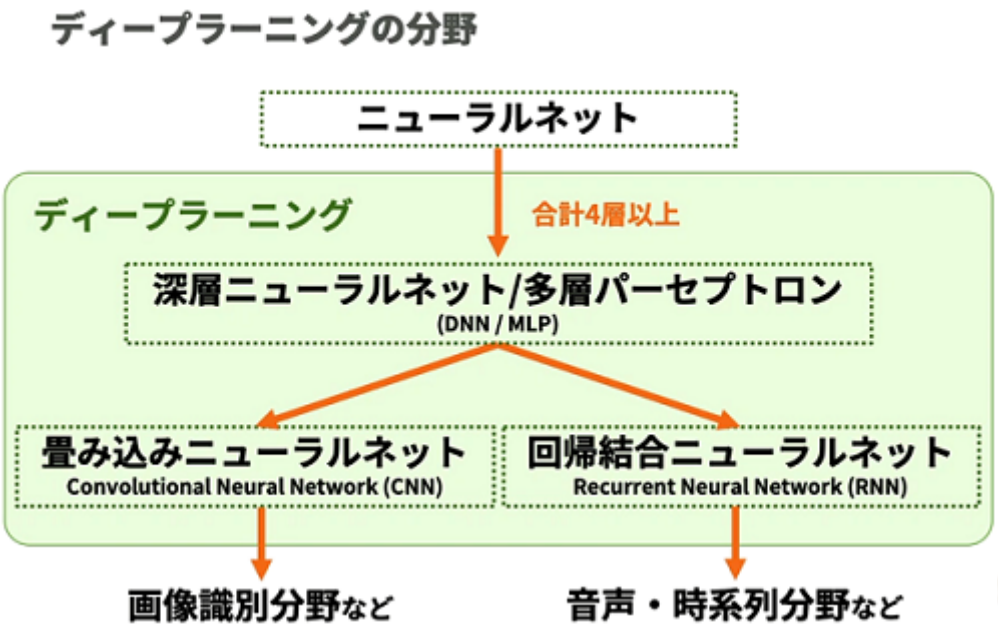
ノード（ニューロン／ユニット）が広くつながったものです。
また、層はレイヤーとも呼ばれています。



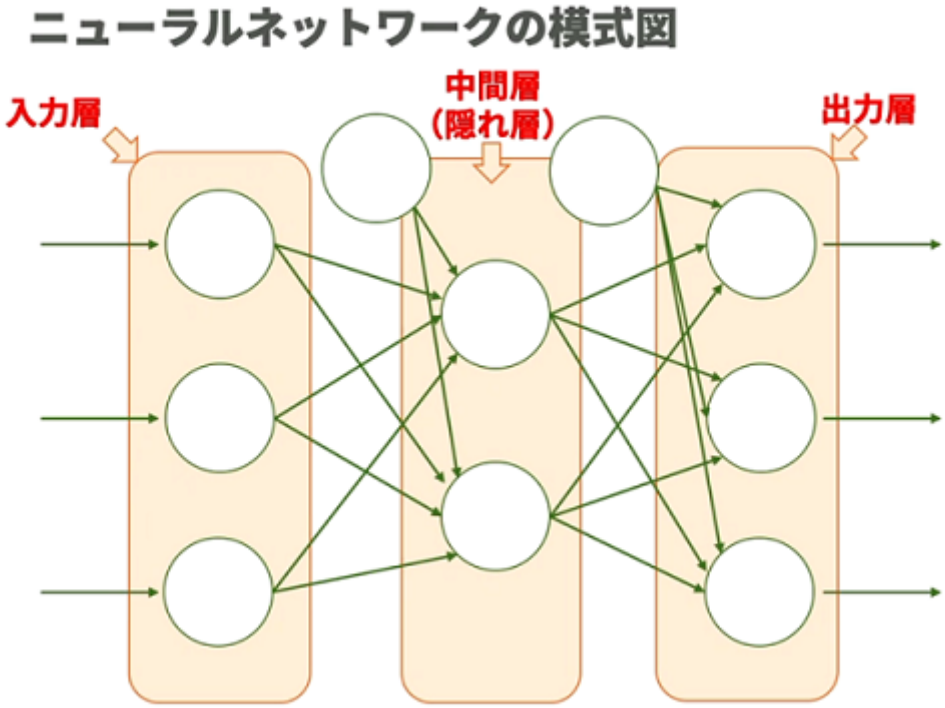
■ ディープラーニング



■ ディープラーニングの定義 ： ニューラルネットワーク（NN）が4層以上（深い層）のもの = 深層学習

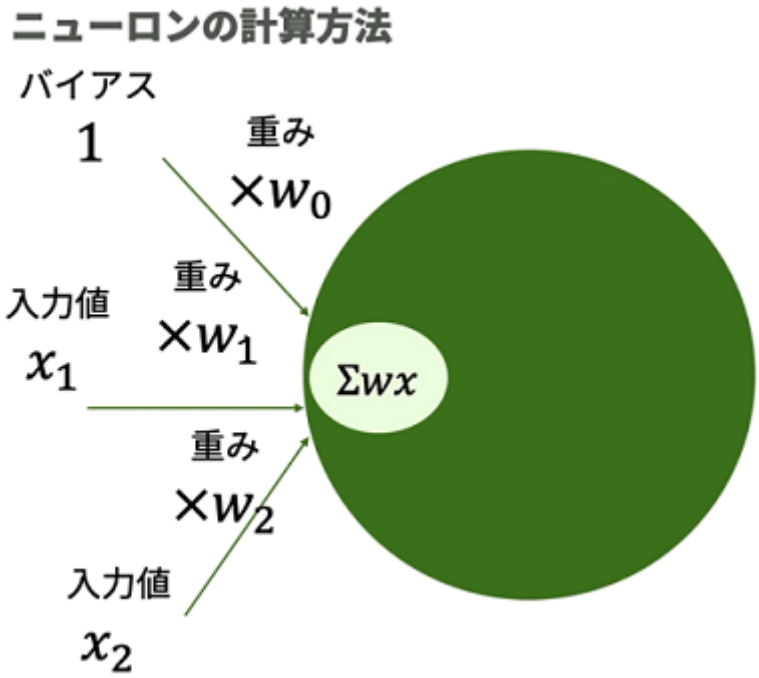


■ ニューラルネットワーク（NN）の構造



■ ニューロンの計算方法

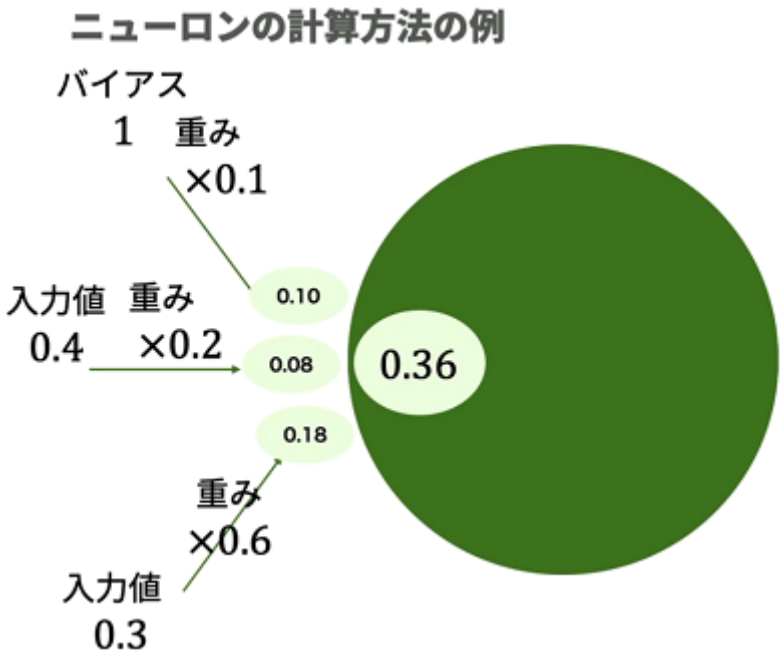
1つのノードに着目をしてみます。



ここで、具体的な数字を入れてみます。

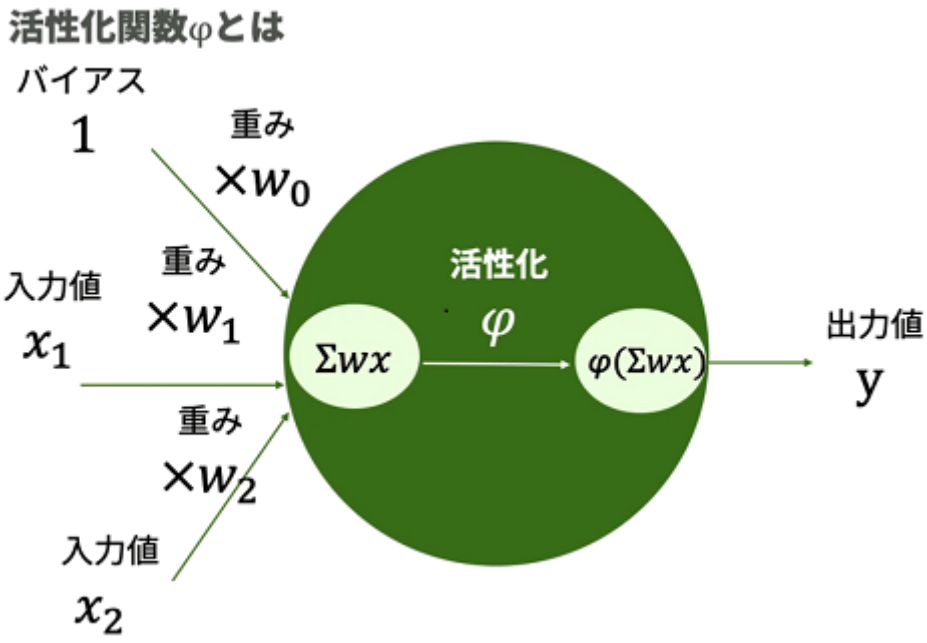
そうすると、見覚えのある式が出てきます。

$$z = \sum wx = w_0 + w_1 x_1 + w_2 x_2$$



ここで、ニューロン内では活性化関数 φ によって数値の変換処理が行われます。

イメージとしては、より数値表現を高めるためのものとなります。

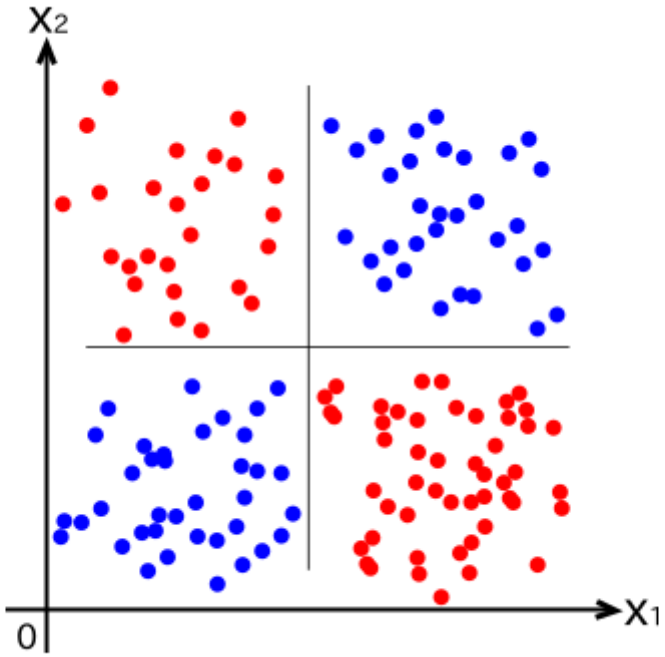


■ 活性化関数とは？

活性化関数とは、全結合層などの後に適用する関数で、ニューロンの**発火**に相当するものです。

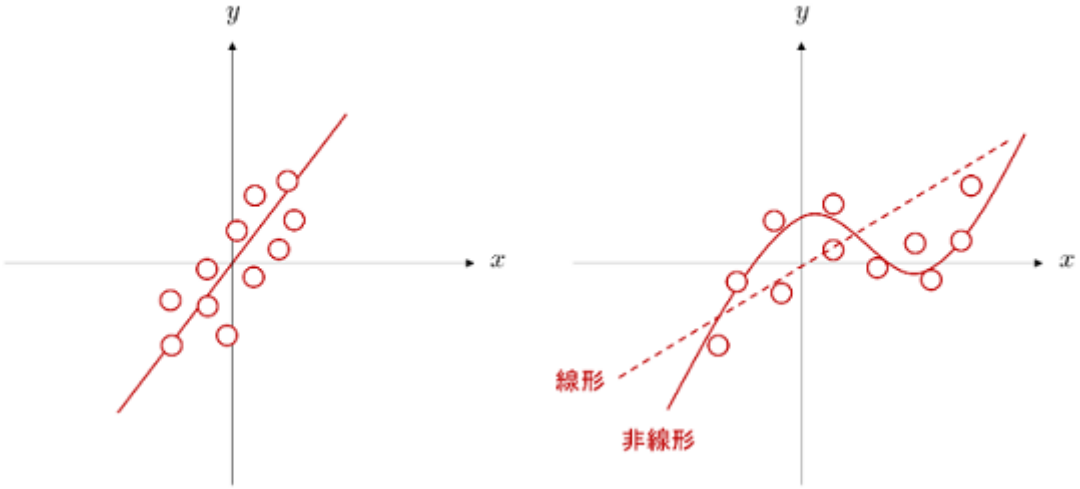
全結合層では、入力を線形変換して出力しますが、**活性化関数を用いることで非線形性をもたせることができます**。

活性化関数を使用しないと、下図のように**一本の直線で分離できず（線形分離不可能）、データが分類できなくなる**のです。



活性化関数によって非線形性をもたせることで、**線形分離不可能なモデルでも、適切に学習が進めば必ず分類できるようになります**。

なお、活性化関数にもいくつかの種類があるので、適切なものを選ぶことが大切です。



■ 活性化関数の種類

活性化関数には数種類があります。

sigmoid関数

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

ReLU関数

$$\varphi(x) = max(0, x) = \begin{cases} 0 & (x \leq 0) \\ x & (x > 0) \end{cases}$$

softmax関数

$$\psi(x) = y(x_k) = \frac{\exp(x_k)}{\sum_{i=1}^{k} \exp(x_i)}$$

中間層ではsigmoid・ReLu、出力層ではsigmoid・softmaxを使用します。

実際にグラフを確認します。

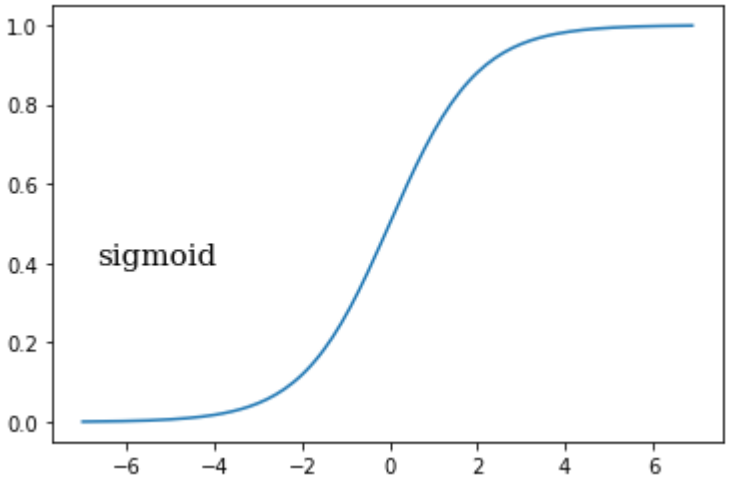
・sigmoid関数

In [46]:

```
# 関数を作成するために、numpyをインポート
import numpy as np

# グラフ描画のために、matplotlibをインポート
import matplotlib.pyplot as plt
```

In [47]:

```
# シグモイド関数を作成
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# x軸の範囲を設定
x = np.arange(-7.0, 7.0, 0.1)

# 出力yの設定
y = sigmoid(x)

# subplot関数の作成
# fig : 描画オブジェクト (= 方眼紙)
# ax : figの中にあるサブプロットオブジェクト (= グラフ自体)
fig, ax = plt.subplots()

# テキストを入力
ax.text(-6.7, 0.4, 'sigmoid', size = 15, family = 'serif')

# 出力プロットの設定
ax.plot(x, y, label = 'sigmoid')

# プロット
plt.show()
```



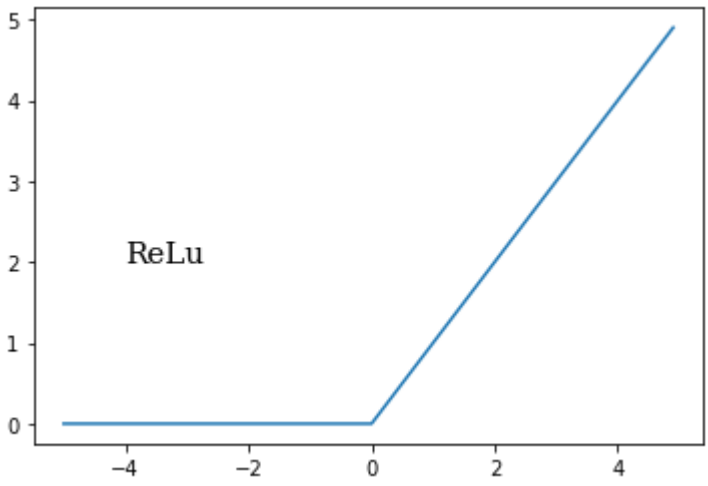
・ReLu関数

In [48]:

```
# relu関数の作成
def ReLU(x):
    y = np.maximum(0, x)
    return y

# -8から8までの値0.1刻みで生成
x = np.arange(-5.0, 5.0, 0.1)

# Xの値をReLU関数へ渡す
y = ReLU(x)

# subplot関数の作成
fig, ax = plt.subplots()

# テキストを入力
ax.text(-4.0, 2.0, 'ReLu', size = 15, family = 'serif')

#プロット
plt.plot(x,y)
```

Out[48]:

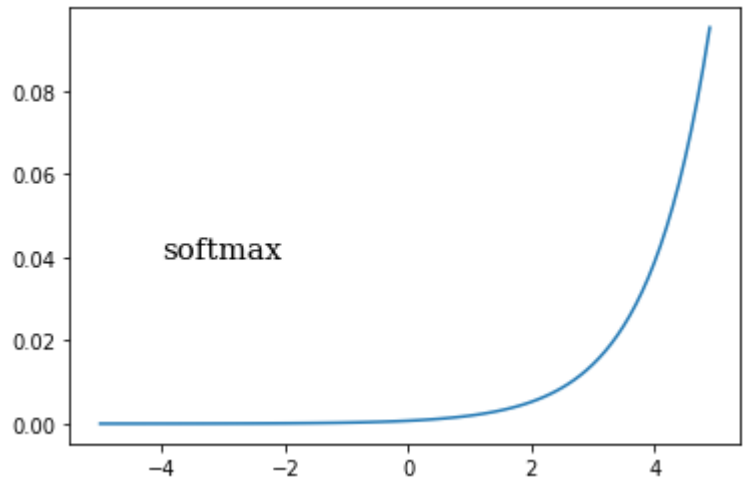
[<matplotlib.lines.Line2D at 0x281977eb188>]



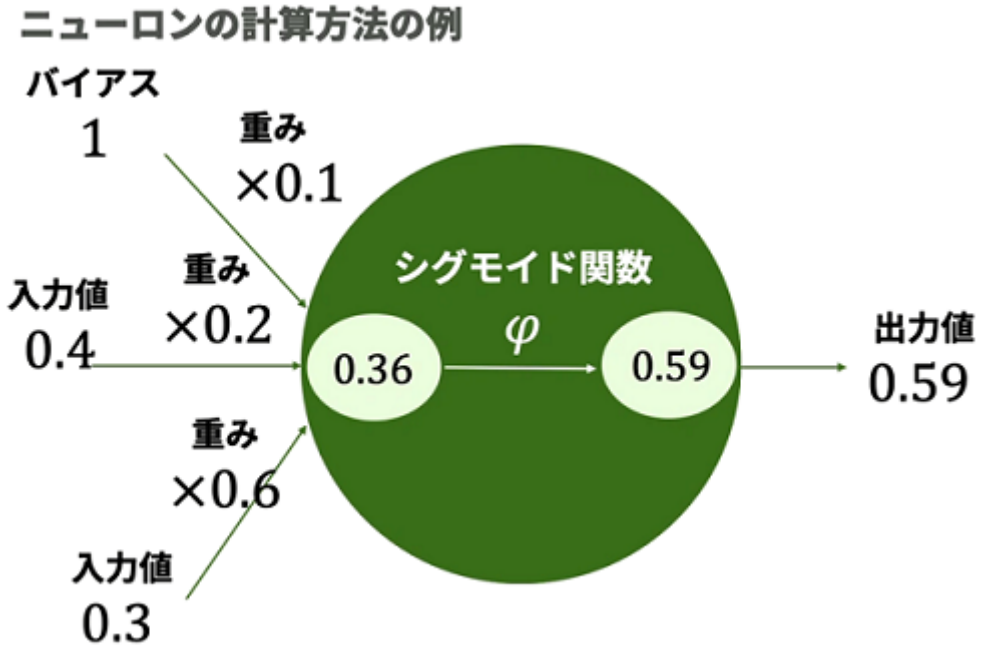
・softmax関数

In [49]:

```
# インスタンスの作成（softmax関数）
def softmax(a):
  exp_a = np.exp(a)
  sum_exp_a = np.sum(exp_a)
  y = exp_a / sum_exp_a
  return y

# xの値をランダムに与える
x = np.arange(-5, 5, 0.1)

# 目的関数yに引数xを与える
y = softmax(x)

# subplot関数の作成
fig, ax = plt.subplots()

# テキストを入力
ax.text(-4.0, 0.04, 'softmax', size = 15, family = 'serif')

# プロット
plt.plot(x, y)
plt.show()
```

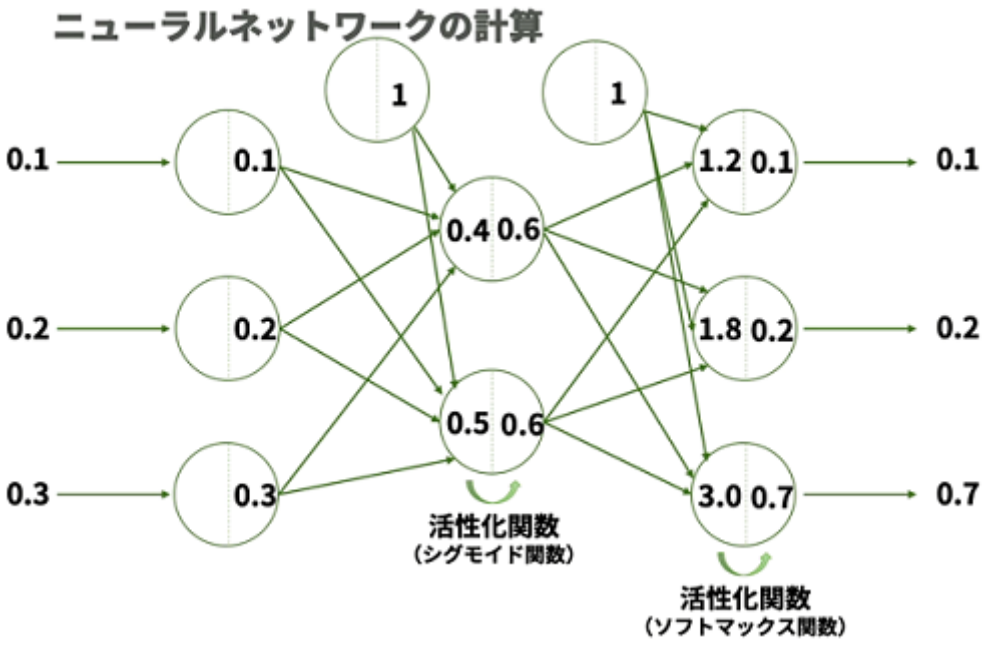


ニューロンの話に戻ります。



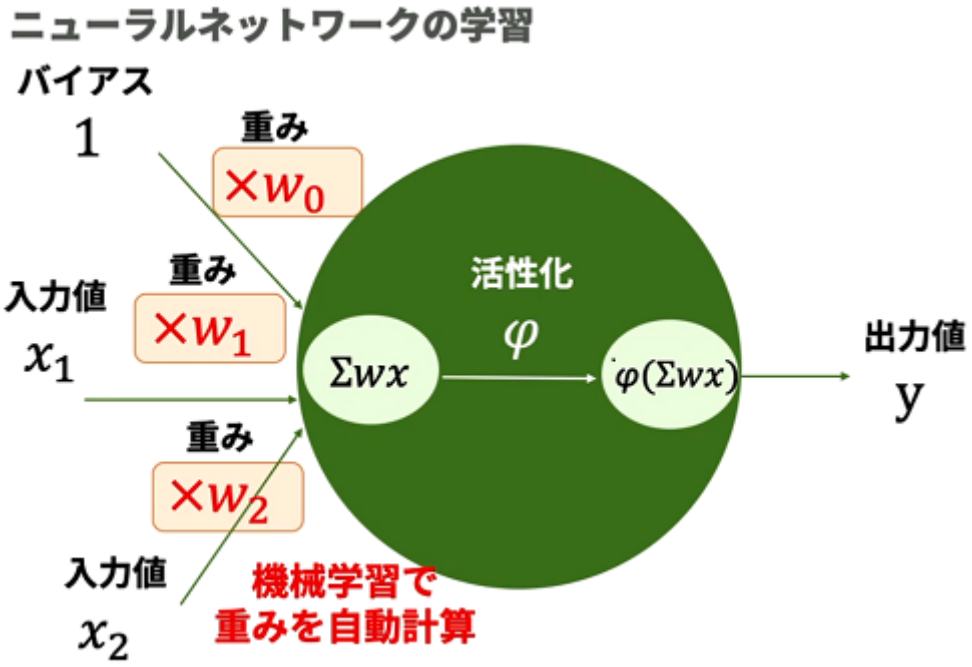
■ニューラルネットワークの計算方法

これを全体（ニューラルネットワーク）にも適用させます。



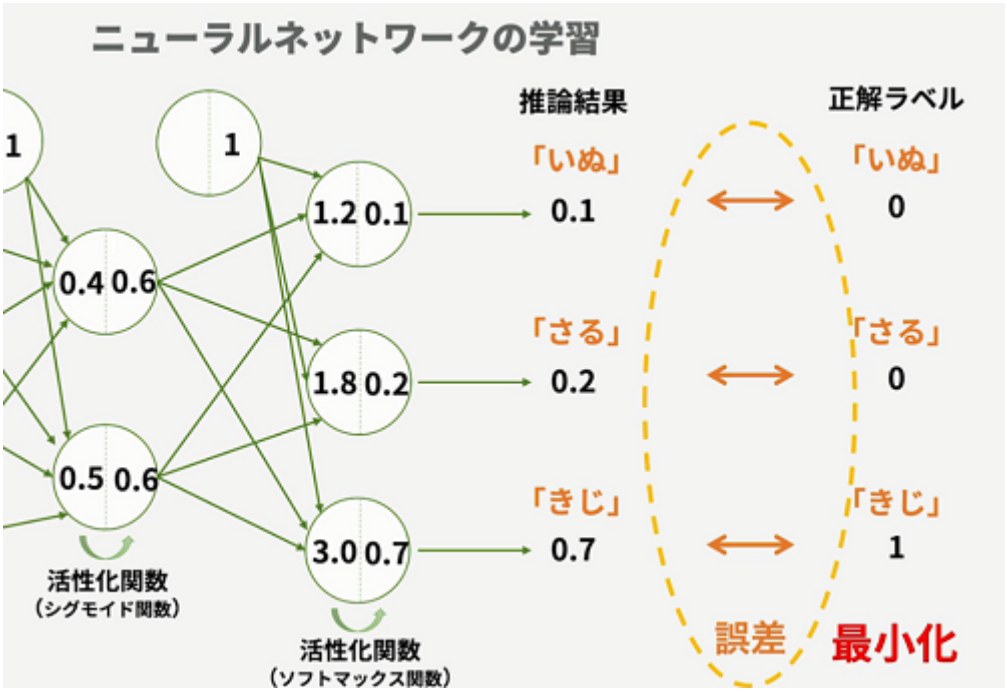
ソフトマックス関数は、出力結果の全体の確率が1になるように変換を行います。

$0.1 + 0.2 + 0.7 = 1$

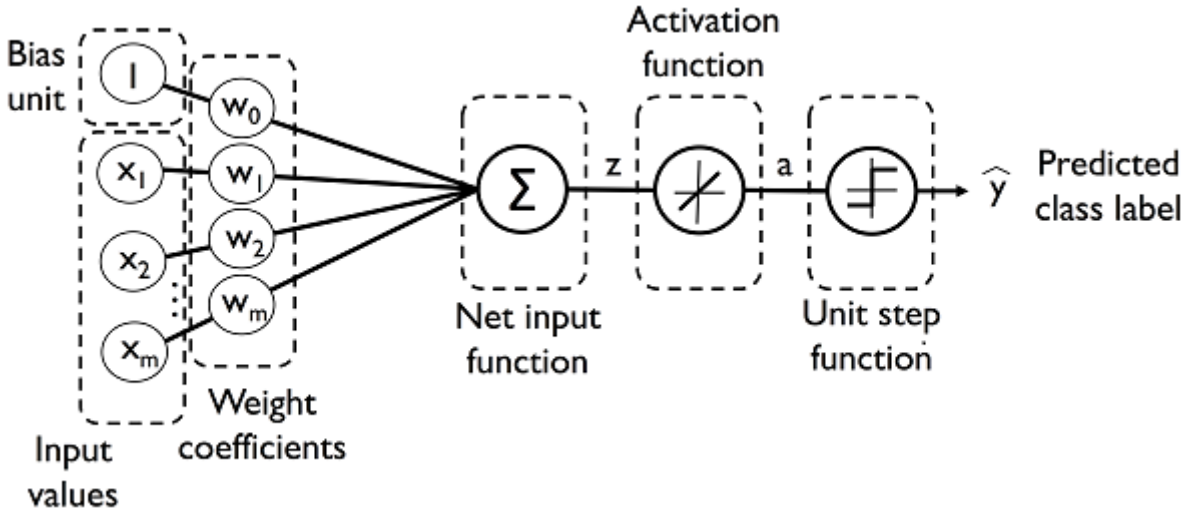


ニューラルネットワークの出力（推論）結果と正解ラベルを比較してみます。

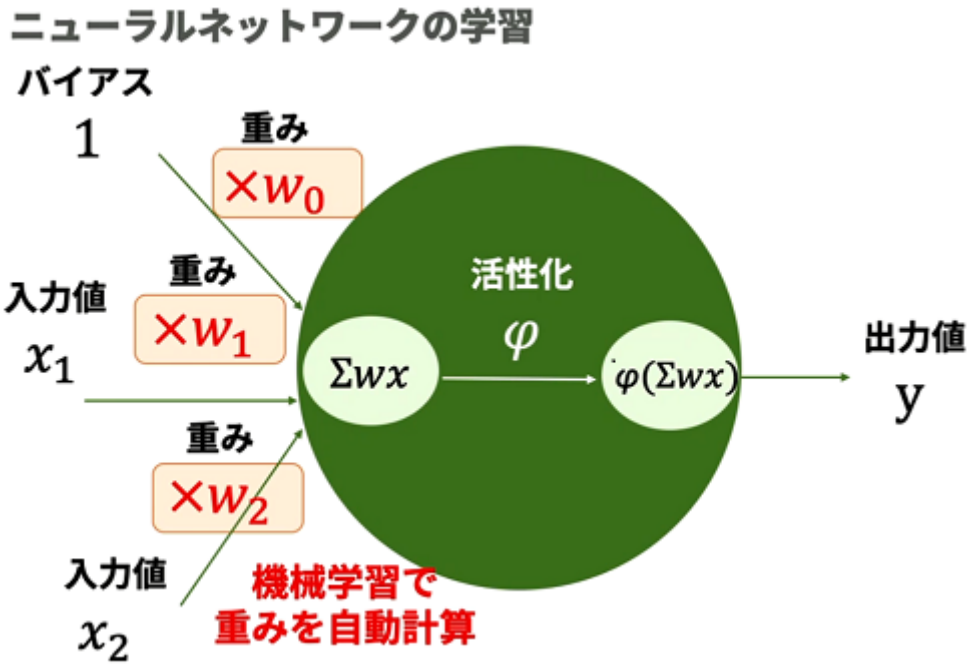
y_predとy_testの誤差を表しており、この損失関数を最小化するように、重みの調整をします。



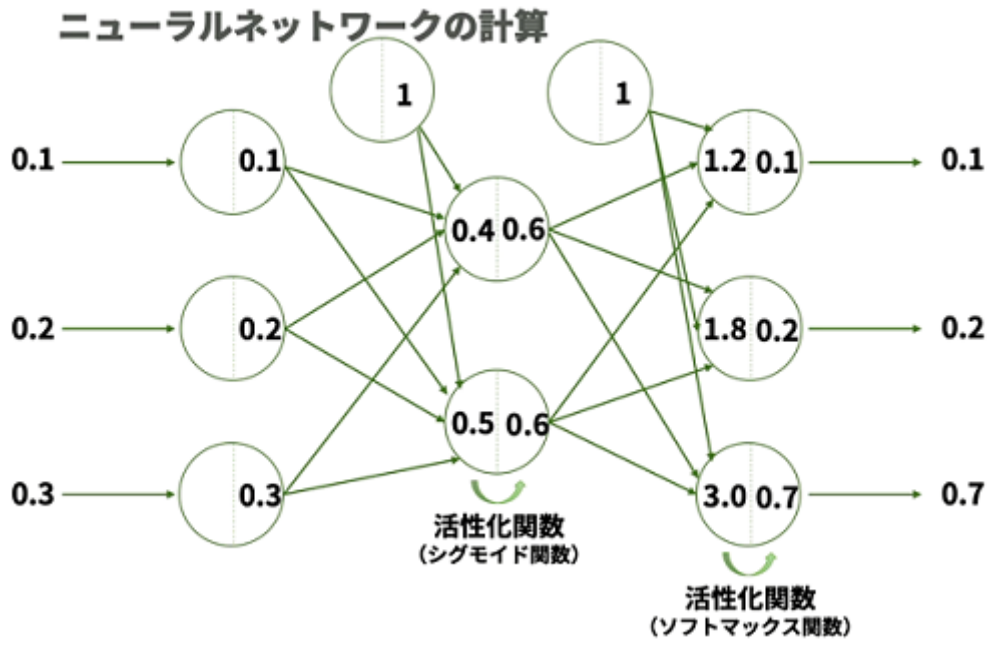
12.1.1　単層ニューラルネットワークのまとめ



上記だと少し難しく見えるが
先ほど説明した通りの内容と同じである。

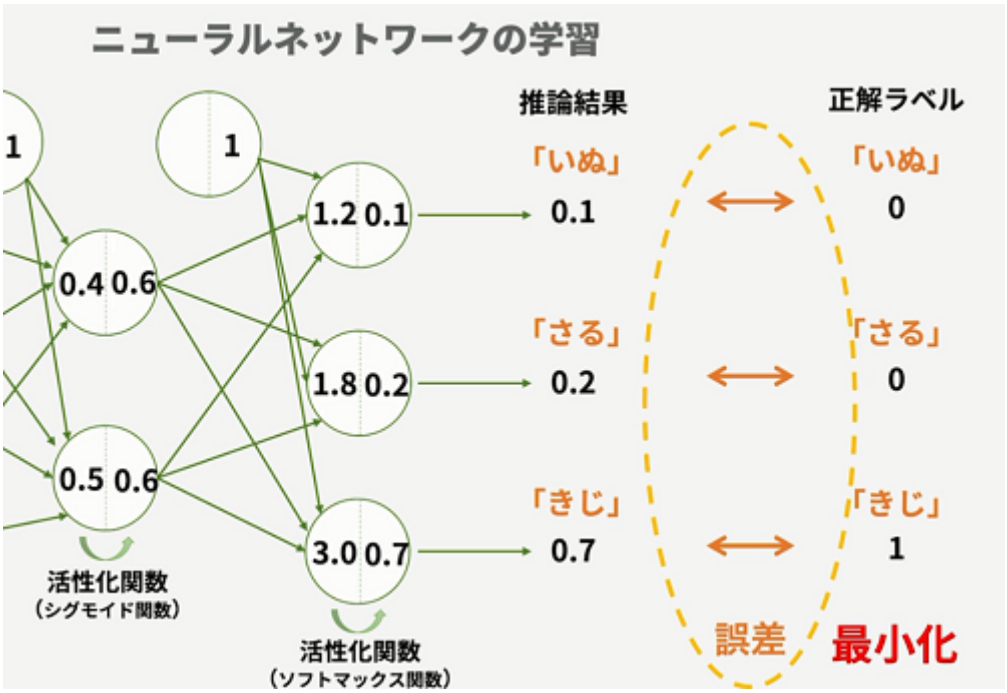


上図を単層パーセプトロン、下図を多層パーセプトロンと呼ぶ。



■ コスト関数の勾配降下法

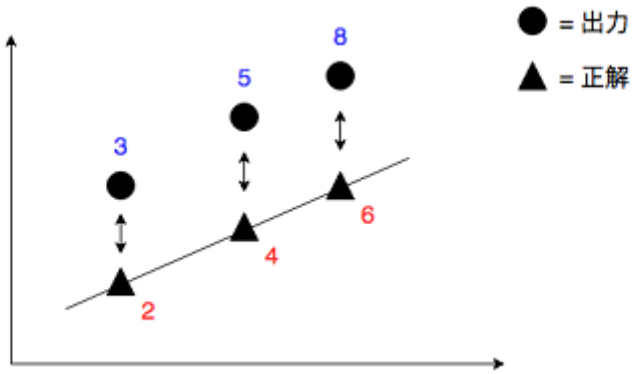
・予測値と出力値の誤差を計算する



・平均二乗誤差

平均二乗誤差は最小二乗法と並んで、統計学などの分野でよく使用される誤差関数です。 下記の式で記述します。

$$E = \frac{1}{N}\sum_{i=1}^{N}(t_i - y_i)^2$$



$$\frac{1}{3}\{(2-3)^2 + (4-5)^2 + (6-8)^2\} = 2$$

図2.5.2-1　平均二乗誤差

y_iは **予測ラベル** 、t_iは **正解ラベル** です。平均二乗誤差は連続値の評価を得意とするため、 **主に回帰モデルの誤差関数** として使われます。
$\frac{1}{N}$はデータ数で決まる定数なので、省略して単に誤差の2乗値を合計した誤差関数を使うこともあります。

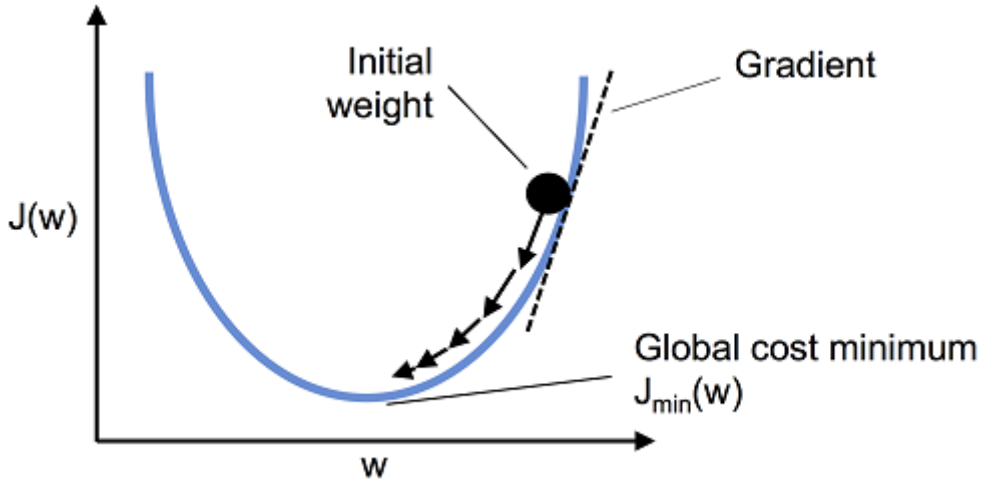
・コスト関数

ニューラルネットワーク全体の重みとしきい値をランダムで決めたときに
ニューロンから出た答えが、正解とどれくらい離れているかを計測するための関数（平均二乗誤差）です。
この関数の値をゼロに近づけていくことが、学習にとって重要となります。

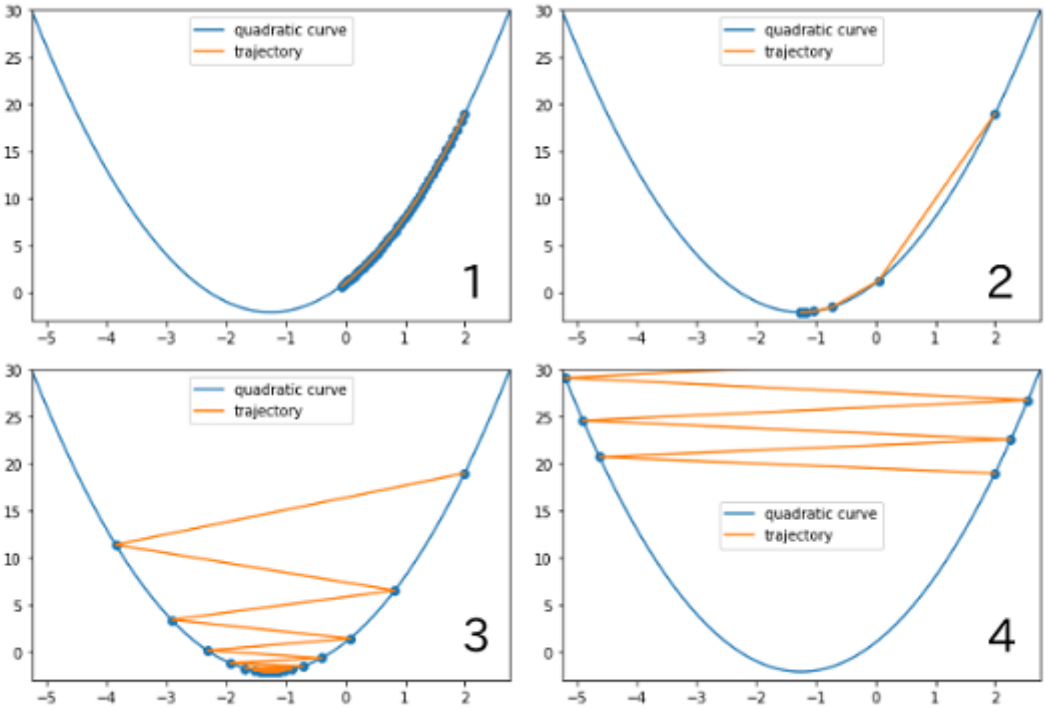
$$J(\mathbf{w}) = \frac{1}{2}\sum_{i}(y^{(i)} - \phi(z^{(i)}))^2$$

尚、$\frac{1}{2}$は計算しやすいように付けています。

イメージとしては、勾配（傾斜面）を少しずつ降りていき、最小値（極小値）に向かう感じです。

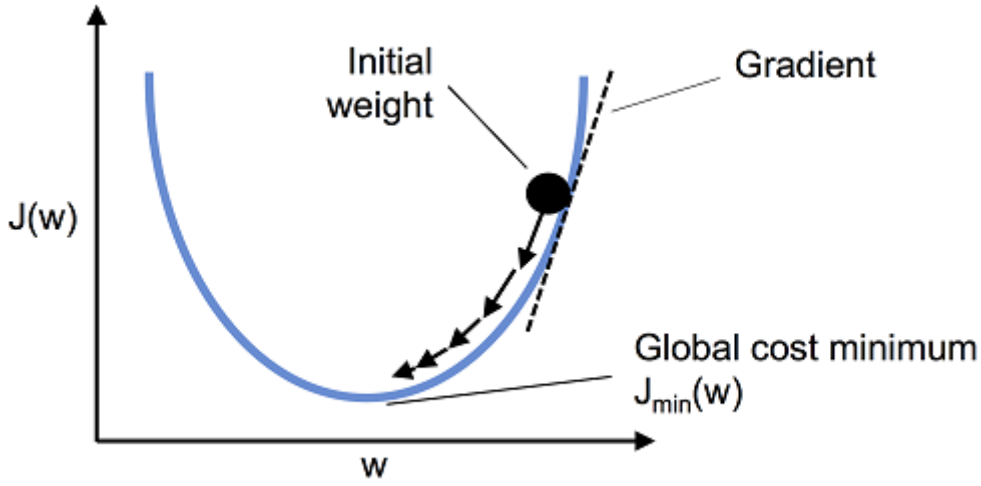


学習率とは、 重みを一度にどの程度変更するかを決めるハイパーパラメータ（手動で調整する値）です。



1. 学習率が低すぎて、ほとんど更新が進んでいません。
2. 適切な学習率により、少ない回数で値が収束しています。
3. 収束はしますが、値が大きいため、更新に無駄があります。
4. 学習率が高すぎて、値が発散しています。(上側に更新され、値がどんどん大きくなっています)

つまり、モデルの学習を適切に行うためには、 **損失関数に対する適切な学習率を設定する必要** があるのです。



ここで最小値に向かう際に、重みを更新（変化）し続けることになるので

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$$

:=：等しいという意味に加えて、右辺を左辺に代入する
∇：偏微分の記号（通常微分でいうΔ）

今、$J(\mathbf{w})$が下げれば$\Delta\mathbf{w}$が上がり、$J(\mathbf{w})$が上がれば$\Delta\mathbf{w}$が下がることから
負の勾配に学習率ηを掛けたものとして下記のように定義することができる。

$$\Delta\mathbf{w} = -\eta\nabla J(\mathbf{w})$$

ここで最初の式のコスト関数を偏微分して、$\nabla J(\mathbf{w})$ を求めていく。

$$J(\mathbf{w}) = \frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

このことから

$$\nabla J(\mathbf{w}) = \frac{\partial J(\mathbf{w})}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

$\frac{1}{2}$を前に出して

$$= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

w_j について合成微分をして

$$= \frac{1}{2}\sum_i 2(y^{(i)} - \phi(z^{(i)}))\frac{\partial}{\partial w_j}(y^{(i)} - \phi(z^{(i)}))$$

$\phi(z^{(i)})$ の中身を展開して

$$= (y^{(i)} - \phi(z^{(i)}))\frac{\partial}{\partial w_j}(y^{(i)} - \sum_i (w_k x_k^{(i)}))$$

j 番目の $w_j x_j^{(i)}$ だけが微分されて

$$= \sum_i (y^{(i)} - \phi(z^{(i)})(-x_j^{(i)})$$

マイナスを外に出して

$$= -\sum_i (y^{(i)} - \phi(z^{(i)})(x_j^{(i)})$$

$\phi(z)$ は、出力値 a と等しいから

$$a^{(i)} = \phi(z)^{(i)}$$

代入して

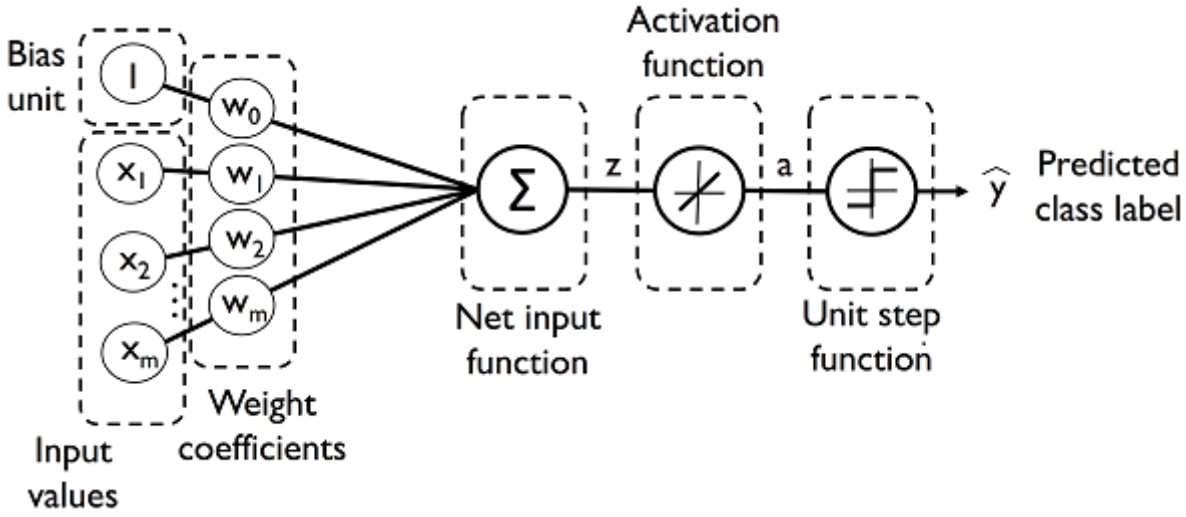
$$\nabla J(\mathbf{w}) = \frac{\partial J(\mathbf{w})}{\partial w_j} = -\sum_i (y^{(i)} - a^{(i)})x_j^{(i)}$$

$z^{(i)}$ ：入力値

$x^{(i)}$：入力値 $z^{(i)}$ の説明変数
$a^{(i)}$：出力値
$y^{(i)}$：予測値（最終的な出力値）

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

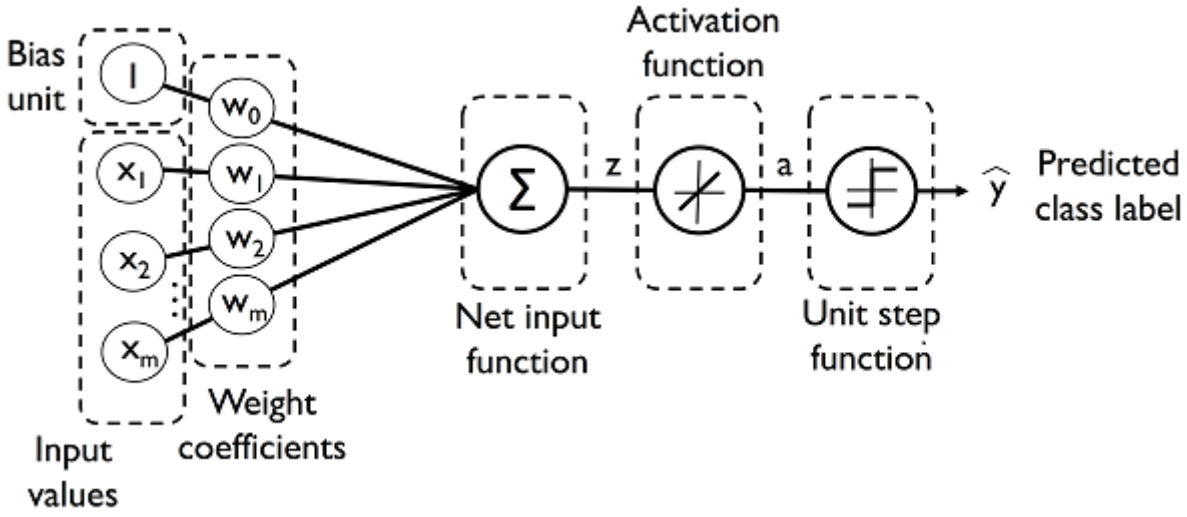
となり、$J(\mathbf{w})$が全ての重み（$w_0 \sim w_i$）によって偏微分されたもの（偏導関数）が求まった。



これにより、コスト関数$J(\mathbf{w})$を最小にする式が求められた。

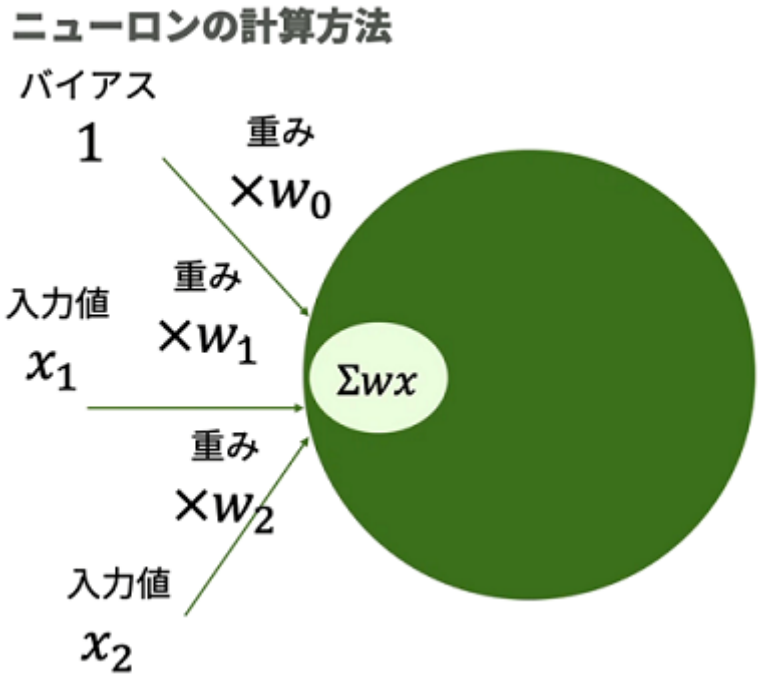
■ 単層ニューラルネットワークの予測値計算

ここで再度、単層ニューラルネットワークのモデルを見てみます。



z：入力値
a：出力値
$\hat{y}$：予測値（最後の出力値）

次に入力値 z について考える。



そのため入力値 z は、次のように表記できる。

$$z = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_mx_m = \sum_{j=0}^{m} w_jx_j = w^Tx$$

ここで、活性化関数を下記のように定義する。

$$\phi(z) = z = a$$

活性化関数 ϕ を使って勾配の計算をするときは
クラスラベルを予測するための、しきい値関数（ヘビサイド関数）を実装することで
連続値の出力を二値のクラスラベルの予測に振り分けることができる。

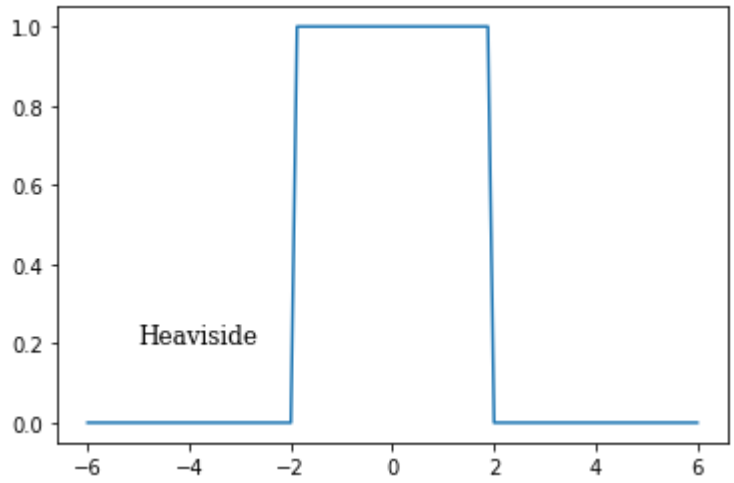
$$\hat{y} = \begin{cases} 1 & (g(z) \geq 0) \\ -1 & (g(z) < 0) \end{cases}$$

・ヘビサイド関数（しきい値関数／ステップ関数）

出力値が0か1の関数のことである。

In [2]:

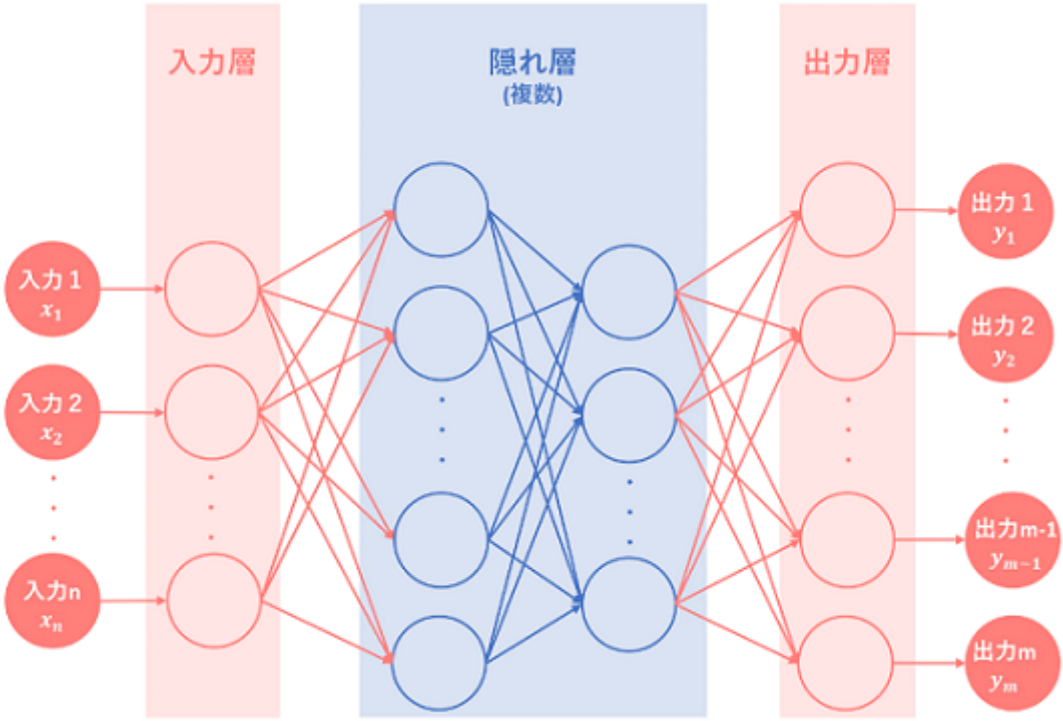
```
# ライブラリ(数値計算、グラフ出力用)
import numpy as np
import matplotlib.pyplot as plt

# ライブラリ(ヘビサイド関数の数式モデル構築用)
from sympy import Heaviside

# 変曲点の設定
f = lambda x: Heaviside(-(x - 2)*(x + 2), 0)

# numpy のベクトルに関数を適用できるように np.vectorize を使用
f = np.vectorize(f)

# xの範囲を指定
xdata = np.linspace(-6, 6, 100)

# 出力値yにxを挿入する
ydata = f(xdata)

# subplot関数の作成
fig, ax = plt.subplots()

# テキストを入力
ax.text(-5.0, 0.2, 'Heaviside', size = 12, family = 'serif')

# xdata, ydata をプロット
plt.plot(xdata, ydata)

# Falseにすると、グラフの形が変わる
plt.show(True)
```



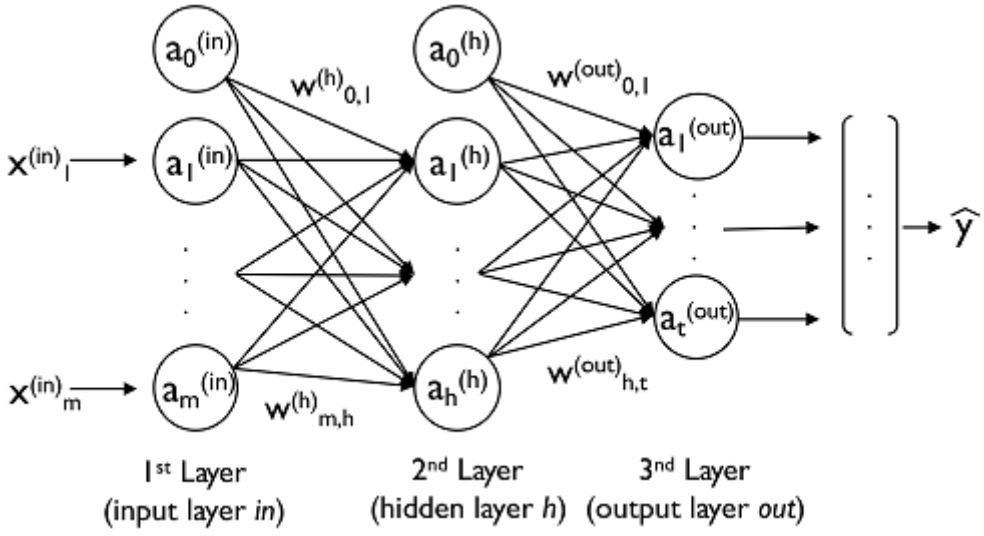
12.1.2　多層ニューラルネットワークアーキテクチャ

■ 多層ニューラルネットワークの構造

ニューラルネットワークを3つの層に分ける。



さらに詳細な情報を記載すると、下図のようになる。



入力層 in
隠れ層 h（中間にある層）
出力層 out

$w_{m,h}$：mからhにかかる重み
a ： xを入力したときの出力値

■ 隠れ層の重みの次元について考える

ここで、第1層（入力層）では、$a_0^{(in)}$はバイアスとなるため、1に設定される。

$$a_0^{(in)} = 1$$

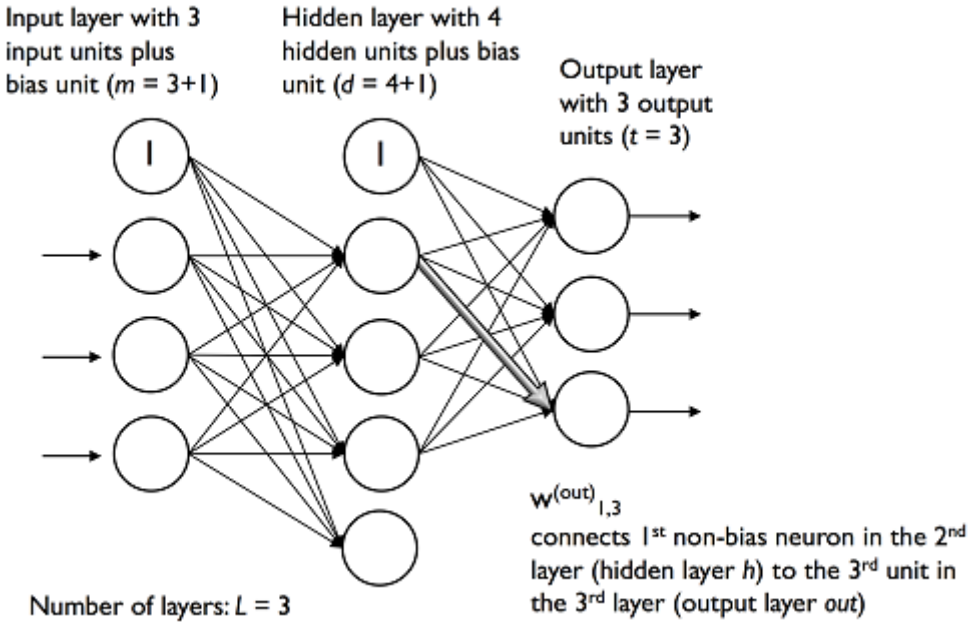
また入力層では、$x_{1\sim n}^{(in)}$は活性化関数で変換せず、そのまま出力される。

$$a_{1\sim n}^{(in)} = \begin{bmatrix} a_1^{(in)} \\ \cdot \\ \cdot \\ \cdot \\ a_n^{(in)} \end{bmatrix} = \begin{bmatrix} x_1^{(in)} \\ \cdot \\ \cdot \\ \cdot \\ x_n^{(in)} \end{bmatrix}$$

よって $a^{(in)}$ 全体では

$$a_{0\sim n}^{(in)} = a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \cdot \\ \cdot \\ \cdot \\ a_n^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \cdot \\ \cdot \\ \cdot \\ x_n^{(in)} \end{bmatrix}$$

ここで重みwについては、1つのノードから次の全ノードへと掛かっている。



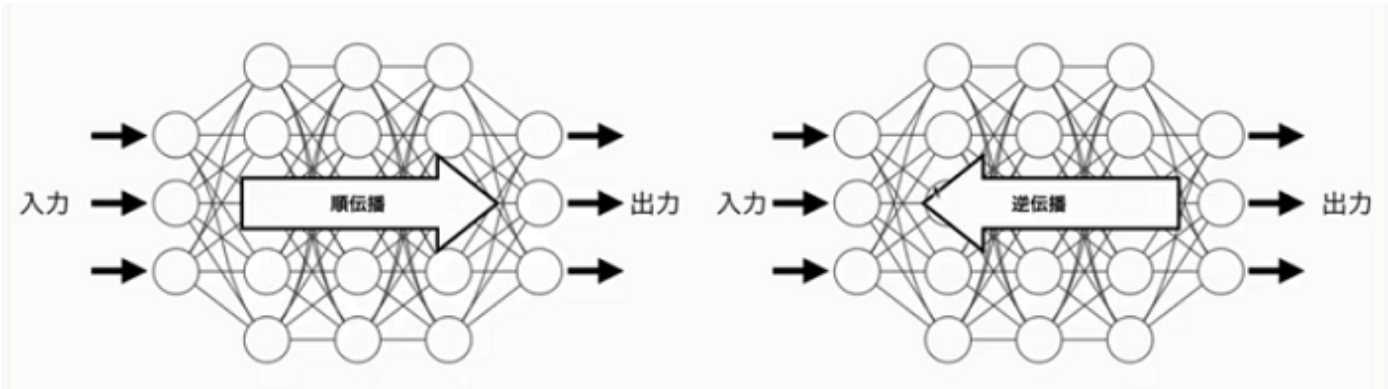
そのため、$\mathbf{W}^{(h)}$は全部で$m \times d$ 通り存在することになる。

12.1.3　フォワードプロパゲージョンのよるNNの活性化

■ 順伝播と逆伝播

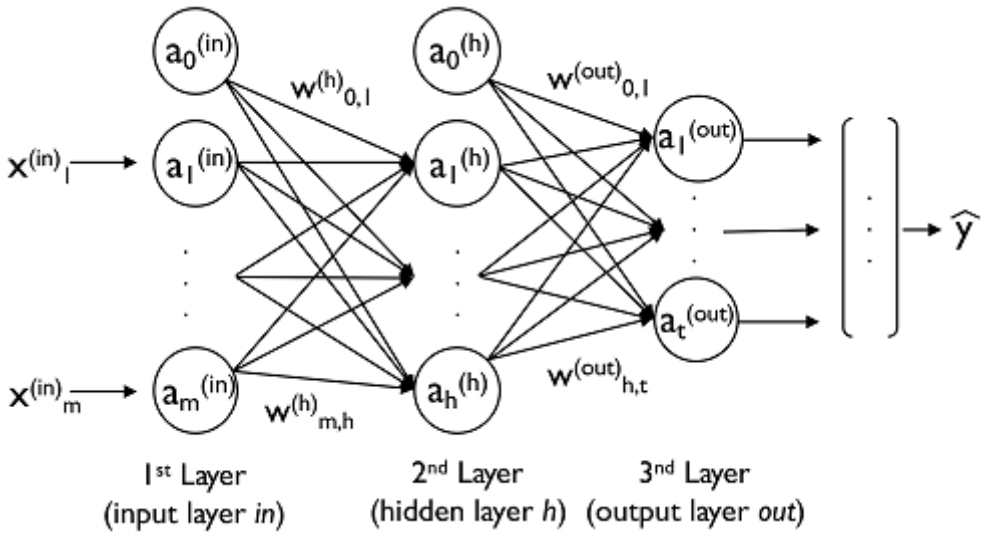
まず重みの調整方法には、順伝播（フォワードプロパゲージョン）と逆伝播（バックフォワードプロパゲージョン）があります。

主な違いとしては、データの重みを更新する方向となります。



■ ディープラーニング（NN）の3ステップ

① これまでのノードの計算方法を用いて、最終的な予測値 $\hat{y}$ を出力する
② 予測値 $\hat{y}$ と正解値 y の誤差（二乗誤差： $J(\mathbf{w})$ ）を求め、それをもとに最小になるような重み w を計算する
③ 求めた重み w を用いて、逆方向に伝播をさせて、NNのモデルを最適なものになるように更新していく



図から、以下の式で表すことができる。

$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \ldots a_m^{(in)} w_{m,1}^{(h)}$$

$$a_1^{(h)} = \phi(z_1^{(h)})$$

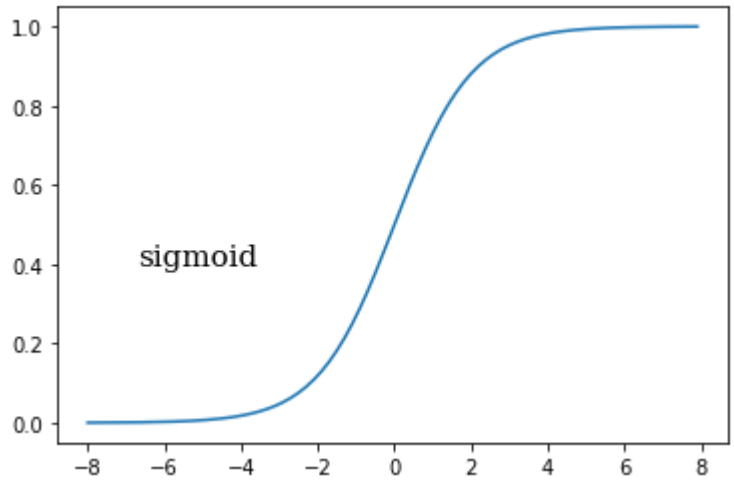
$z_1^{(h)}$ ：総出力
$\phi(\cdot)$：活性化関数

多層パーセプトロンモデルでの手書き文字画像の分類などを行うときは
ロジスティック回帰（シグモイド関数など）を行い、非線形の活性化関数が必要となる。

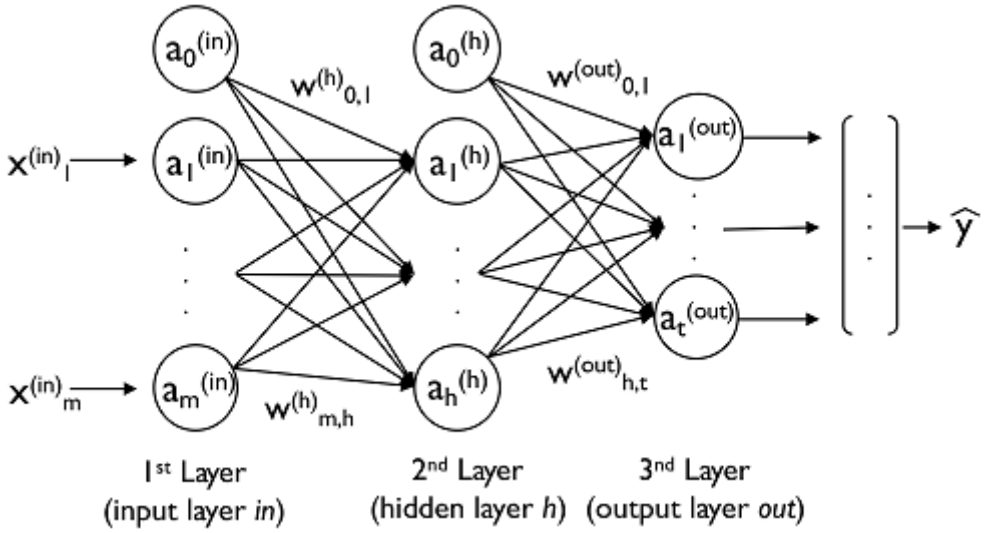
$$\phi(z) = \frac{1}{1 + \exp(-z)}$$

In [54]:

```
# シグモイド関数を作成
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# x軸の範囲を設定
x = np.arange(-8.0, 8.0, 0.1)

# 出力yの設定
y = sigmoid(x)

# subplot関数の作成
# fig : 描画オブジェクト (= 方眼紙)
# ax : figの中にあるサブプロットオブジェクト (= グラフ自体)
fig, ax = plt.subplots()

# テキストを入力
ax.text(-6.7, 0.4, 'sigmoid', size = 15, family = 'serif')

# 出力プロットの設定
ax.plot(x, y, label = 'sigmoid')

# プロット
plt.show()
```



もう一度、図を見ながら式で表していく。



なかなか良い図が見つからなかったので、板書でも書きます。

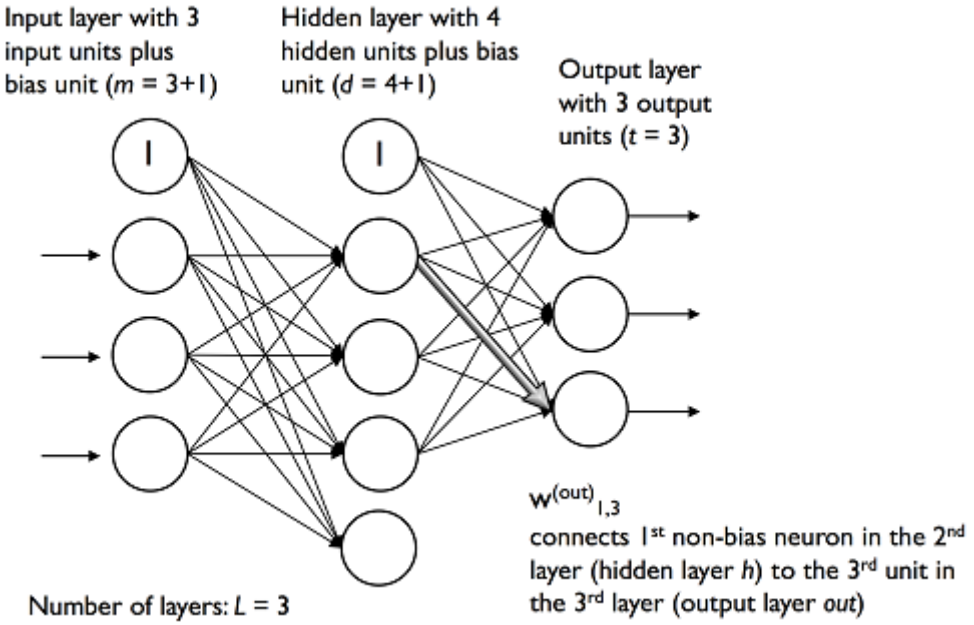
まず隠れ層 (h)に入る、値 $z^{(h)}$ は

$$z^{(h)} = \mathbf{a}^{(in)} W^{(h)}$$

$z^{(h)}$ にシグモイド関数で変換されたものが、 $a^{(h)}$ として出力されるので

$$\mathbf{a}^{(h)} = \phi(z^{(h)})$$

となる。



$\mathbf{a}^{(in)}$：サンプル $x^{(in)}$ にバイアスを足した $1 \times m$ 次元のベクトル
$\mathbf{W}^{(h)}$： $m \times d$ 次元のベクトル
$z^{(h)} = \mathbf{a}^{(in)} W^{(h)}$：$1 \times d$ 次元のベクトル
d：ニューラルネットワーク内の隠れユニットの個数

そして $\mathbf{a}^{(in)}$ のトレーニングデータセットを 1 個から n 個のサンプルに変更する。
つまり、$\mathbf{a}^{(in)}$ を$1 \times m \rightarrow n \times m$ 次元に変換する。

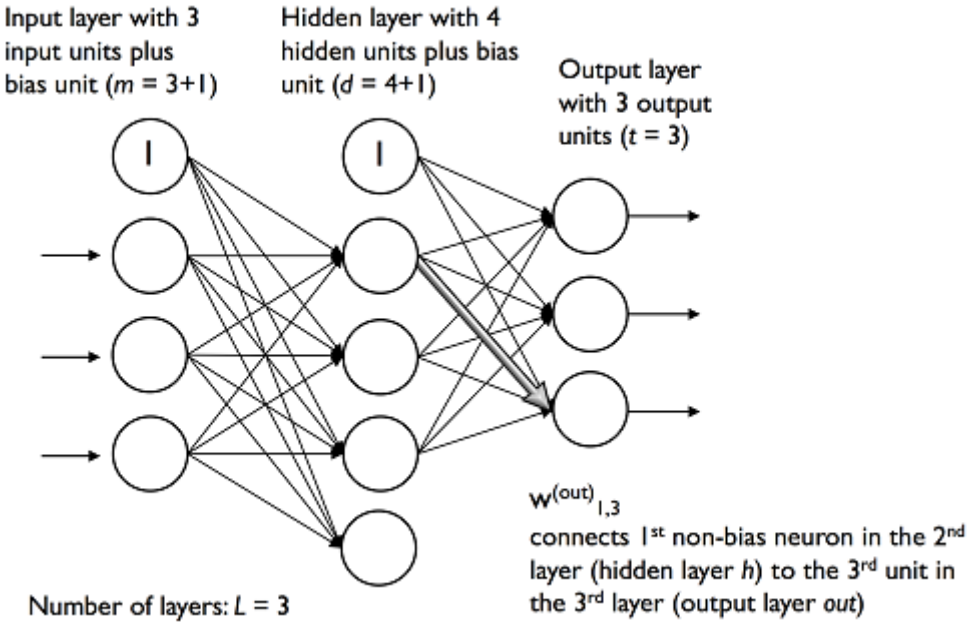
$$Z^{(h)} = \mathbf{A}^{(in)} W^{(h)}$$

$\mathbf{A}^{(in)}$：$n \times m$ 行列
$\mathbf{W}^{(h)}$： $m \times d$ 次元のベクトル
$z^{(h)} = \mathbf{a}^{(in)} W^{(h)}$：$n \times d$ 次元のベクトル

$Z^{(h)}$ を活性化関数で変換して $\mathbf{A}^{(h)}$ が求める。

$$\mathbf{A}^{(h)} = \phi(Z^{(h)})$$

次に、出力層について考える。



$\mathbf{A}^{(h)}$：$n \times d$ 行列
$\mathbf{W}^{(out)}$： $d \times t$ 次元のベクトル
$z^{(out)} = \mathbf{a}^{(in)} W^{(h)}$：$n \times t$ 次元のベクトル

$$Z^{(out)} = \mathbf{A}^{(h)} W^{(out)}$$

最後に$Z^{(out)}$ を活性化関数で変換して $\mathbf{A}^{(out)} = \hat{y}$ が求まる。

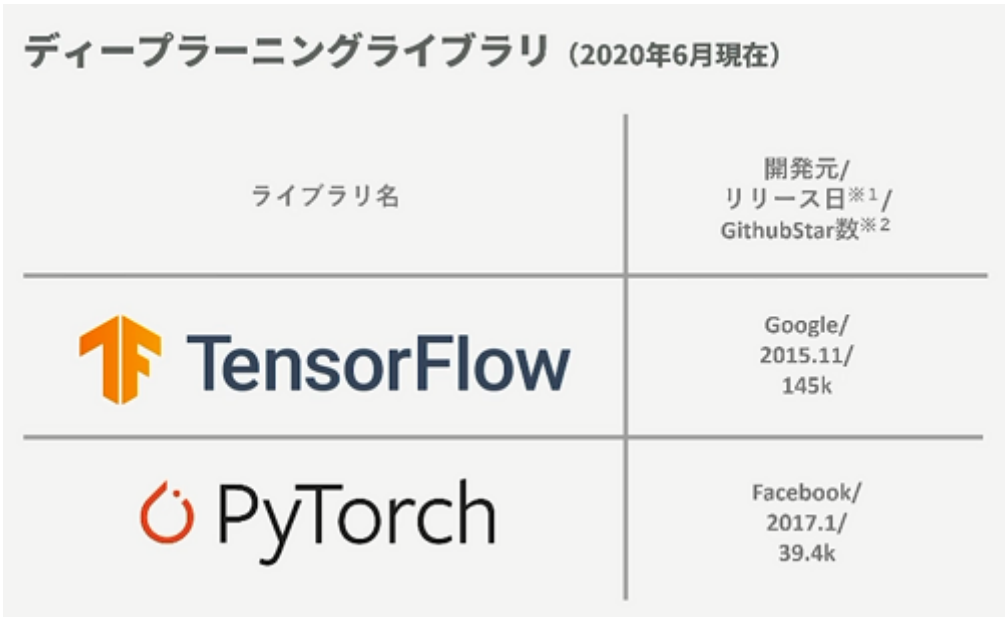
$$\mathbf{A}^{(out)} = \phi(Z^{(out)})$$

$\mathbf{A}^{(out)}$ は $z^{(out)}$：$n \times t$ 次元の関数となっているため
同じく、$n \times t$のベクトルとなる。

参考資料

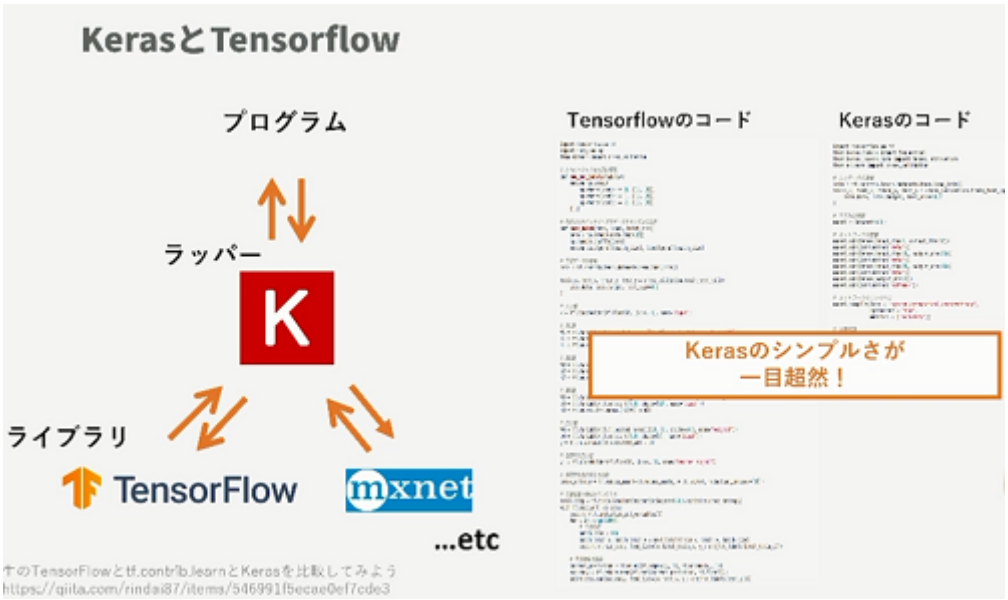
■ ディープラーニング実装の準備（ライブラリ）

次に、ディープラーニングで使用するライブラリについて説明します。
（後ほど、プログラムを実装する際に使用します）

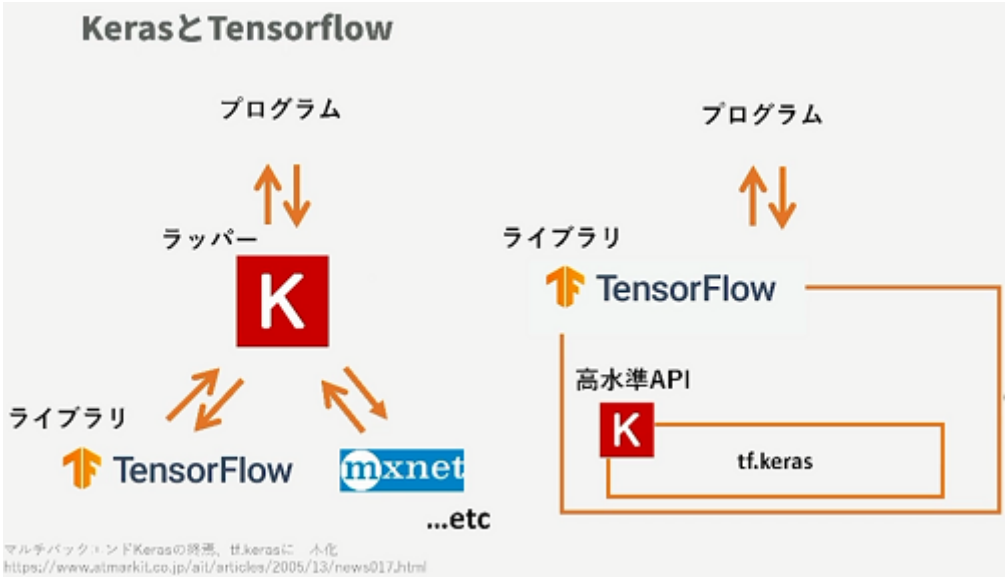


TensorFlow：実際の解析によく用いられる
PyTorch：研究分野などに用いられることが多い

Scikit-learn：ディープラーニング以外での機械学習で使用する



元々はKerasとTensorFlowは別々のものでしたが
現在はTensorFlowの中に一機能として、Kerasが組み込まれています。



念のためですが、高・低水準についても
簡単な補足だけさせていただきます。