ニューラルネットワーク

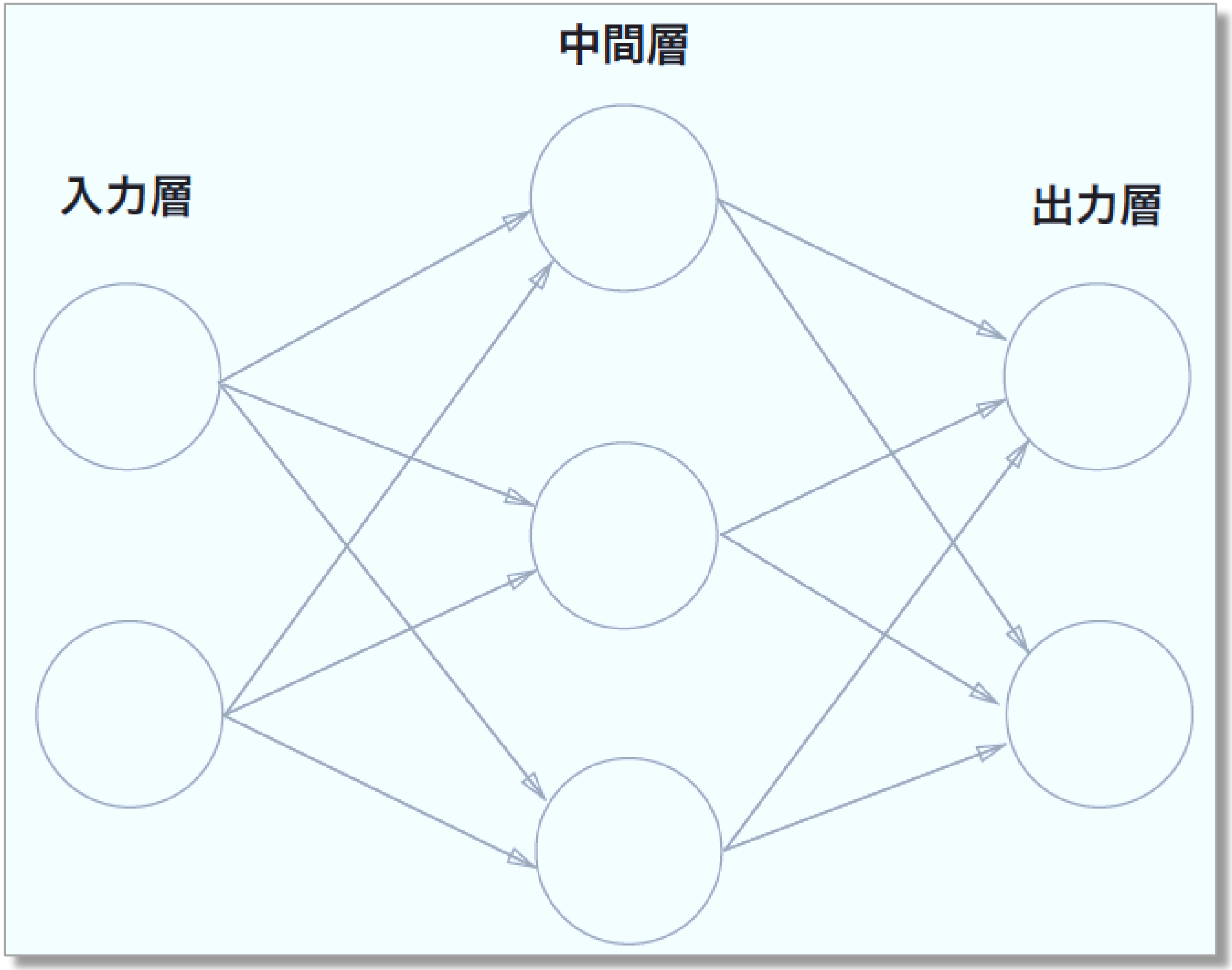
ゼロディープ教科書輪講

2018年5月14日

秋山研究室 修士2年 黄毅聰

目次

ニューラルネットワークの例



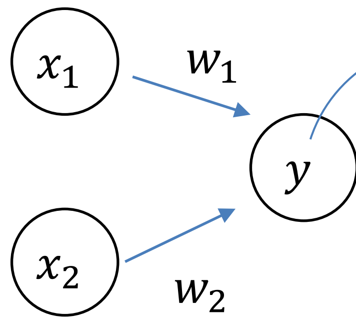
- 中間層は隠れ層と呼ばれることもある
- 本書では、入力層から出力層へ向かって、順に第0層、第１層と呼ぶことにする
- この図では、重みを持つ層が2層なため、本書では「2層ネットワーク」と呼ぶ

パーセプトロンの復習

複数の信号を入力として受け取り、１つの信号(１もしくは0)を出力するアルゴリズム

信号を流すか流さないかの二通りの値

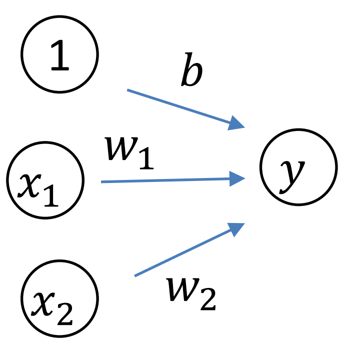
- ○はニューロン
- x_1, x_2は入力信号
- yは出力信号
- w_1, w_2は重み
- bはバイアス



$w_1x_1 + w_2x_2$を受取

$$y = \begin{cases} 0 \ (b + w_1x_1 + w_2x_2 \ \leq 0) \\ 1 \ (b + w_1x_1 + w_2x_2 \ > 0) \end{cases}$$

バイアスを明示的に示すと、



簡略化　　活性化関数 $h(x)$を導入

$$y = h(b + w_1x_1 + w_2x_2)$$

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

目次

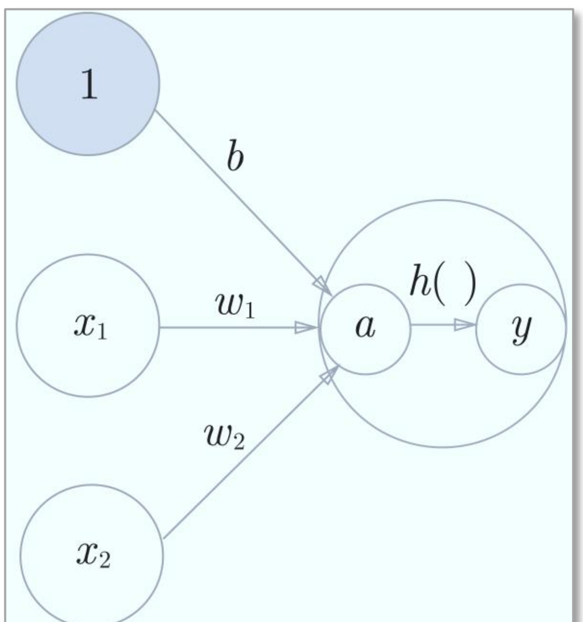
活性化関数の導入

活性化関数：入力信号の総和を出力信号に変換する関数
多層の利点を生かすために非線形関数を使う必要がある

$y = h(b + w_1 x_1 + w_2 x_2)$ を丁寧に書き換えると

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

$$y = h(a)$$



重み付き信号の和の結果が a になり、
活性化関数 h によって y に変換される

活性化関数の例：ステップ関数

ステップ関数：閾値を境にして出力が切り替わる関数

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

実装

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

```
>>> import numpy as np
>>> x = np.array([-1.0, 1.0, 2.0])
>>> x
array([-1.,  1.,  2.])
>>> y = x > 0
>>> y
array([False,  True,  True], dtype=bool)
```

配列のxに対応
できるように

```
>>> y = y.astype(np.int)
>>> y
array([0, 1, 1])
```

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

2行目に、True/Falseという y のブーリアン配列が生成される

3行目にそのブーリアン入配列を0か1に変換

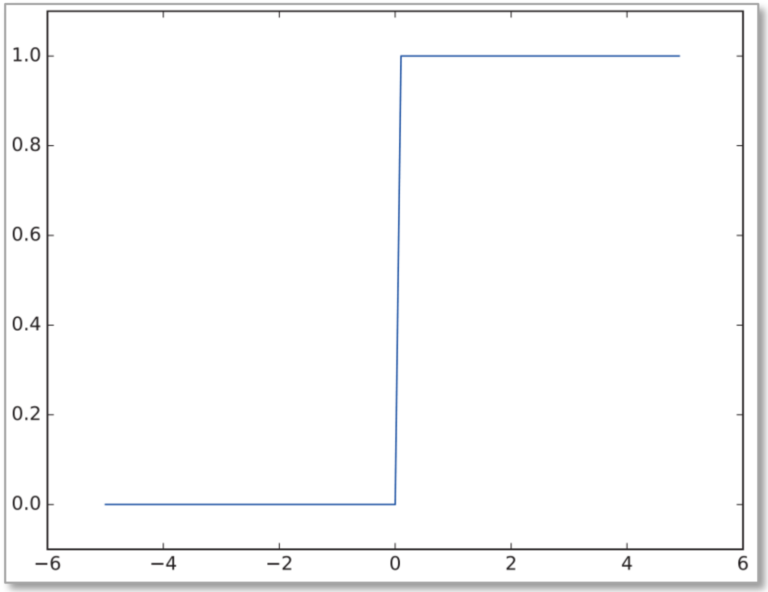
ステップ関数のグラフを表示してみよう

ステップ関数のグラフ


```
import numpy as np
import matplotlib.pylab as plt

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)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # y 軸の範囲を指定
plt.show()
```



活性化関数の例：シグモイド関数

シグモイド関数：

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

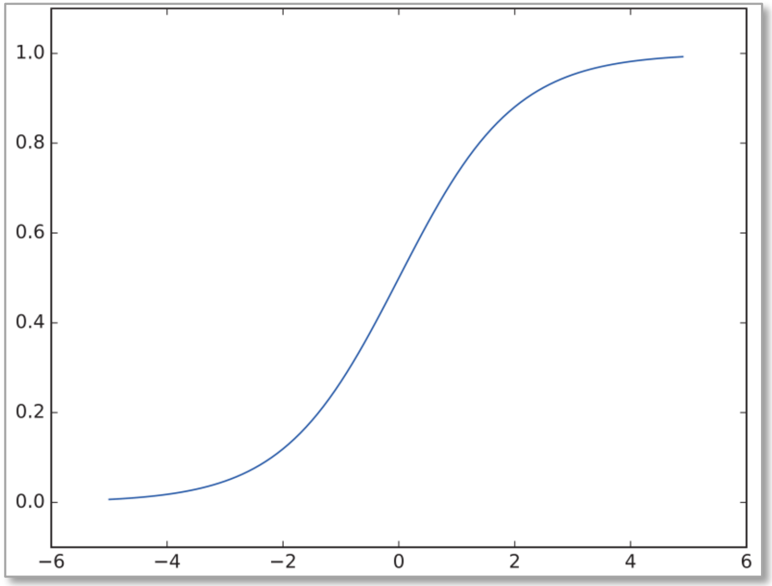
実装

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

ブロードキャストによってスカラー値とNumpy配列の各要素間の演算が可能

グラフ

```
x = np.arange(-5.0, 5.0, 0.1)
y = sigmoid(x)
plt.plot(x, y)
plt.ylim(-0.1, 1.1) # y 軸の範囲を指定
plt.show()
```



シグモイド関数の滑らさがニューラルネットワークの学習において重要

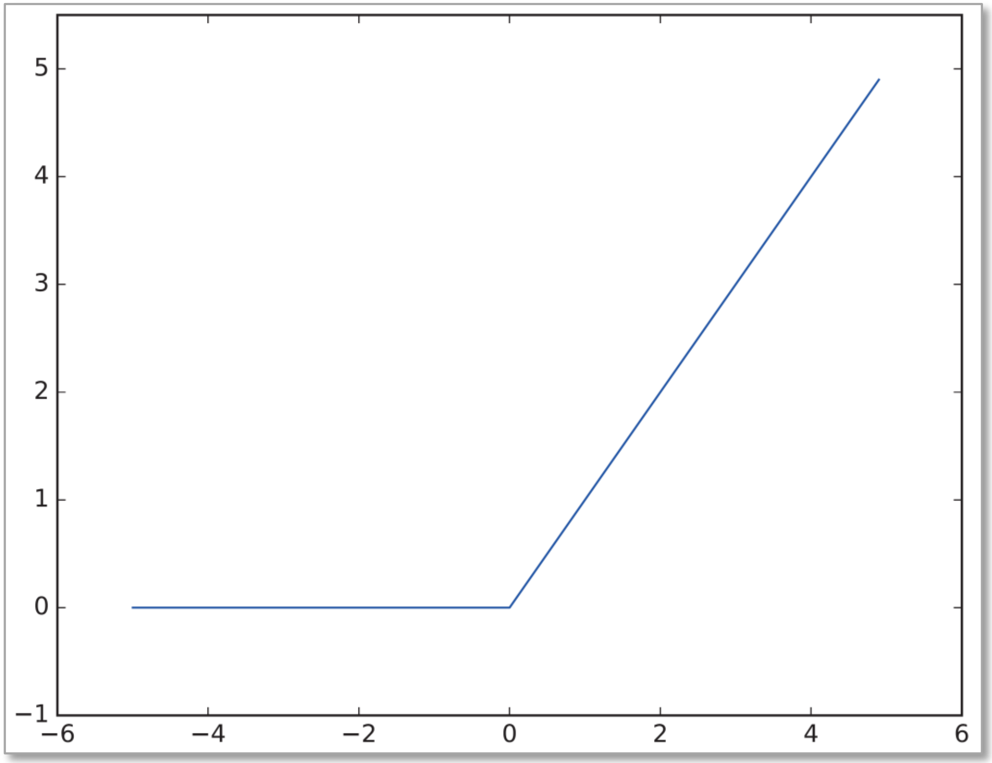
活性化関数の例：ReLU関数

最近よく用いられる関数で、入力が 0 を超えればその入力はそのまま出力し、0 以下ならば 0 を出力

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

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

本の後半はこのReLuを活性化関数として使用



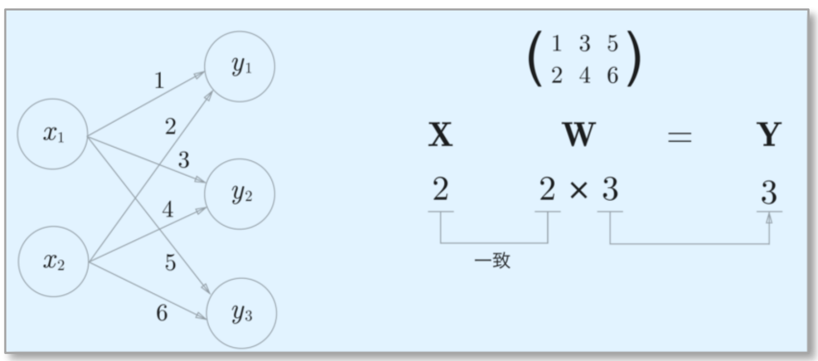
目次

Numpyによる多次元配列の計算

Numpy の多次元配列をマスターすれば、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

- 最初の次元は0番目の次元、次の次元は1番目という順に対応
- Numpyによる行列の内積計算： np.dot(***A***, ***B***) 順番が大事。行列Aの1次元目の要素と行列Bの0次元目の要素は同じでなければ

ニューラルネットワークの行列の積




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

目次

3層ニューラルネットワークの実装

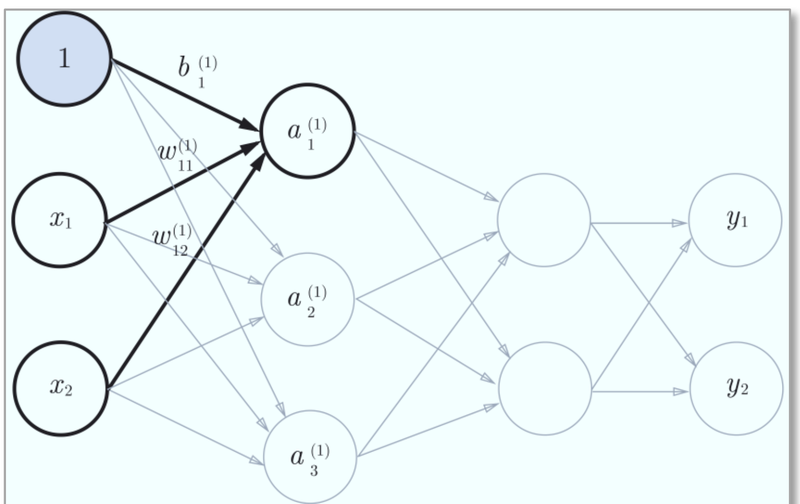
入力層から第1層目への信号の伝達

第1層目の重み

重みの記号：$W_{1\,2}^{(1)}$

前層の2番目のニューロン

次層の1番目のニューロン



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

$$\mathbf{A}^{(1)} = (a_1^{(1)} \ \ a_2^{(1)} \ \ a_3^{(1)}) \qquad X = (x_1 \ \ x_2)$$

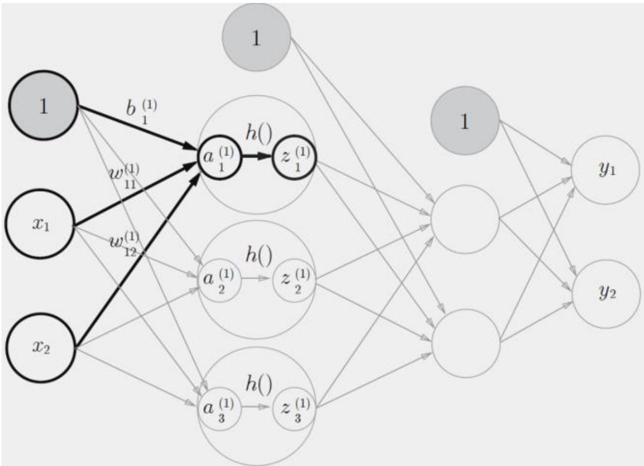
$$W = \begin{pmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{pmatrix}$$

$\mathbf{X}$ $\mathbf{A}^{(1)}$

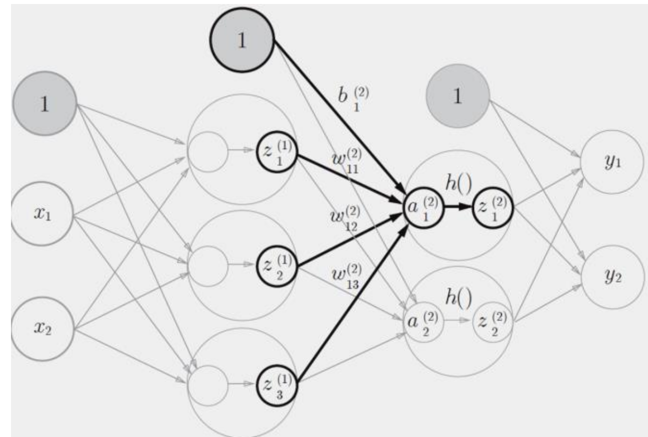
$$B = (b_1^{(1)} \ \ b_2^{(1)} \ \ b_3^{(1)})$$

3層のニューラルネットワークの信号伝達

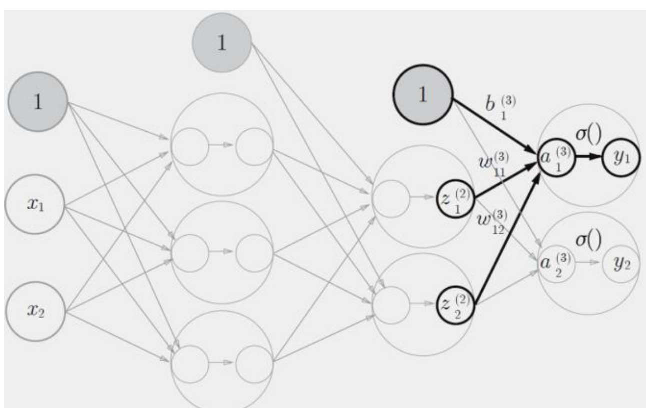
入力層から第1層へ信号の伝達



第1層から第2層へ信号の伝達



第2層から出力層へ信号の伝達



- 活性化関数：シグモイド関数
  >> **Z** = sigmoid(**A**)
- 第1層目から第2層目への信号の伝達において、**Z1**を入力信号として、**A2**を計算し、活性化関数によって**Z2**に変換
- 第2層から出力層への信号の伝達において、**最後の活性化関数だけ隠れ層と異なり**、恒等関数 y = x を使用

実装のまとめ

three_layer_neural_network.py

```python
import numpy as np

#重みとバイアスのパラメーターを辞書型のnetwoekに格納する
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

#ニューラルネットワーク順方向伝播の計算
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 = function_sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        z2 = function_sigmoid(a2)
        a3 = np.dot(z2, W3) + b3
        y = function_identify(a3)

        return y

#シグモイド関数
def function_sigmoid(x):
        return 1/(1+np.exp(-x))

#恒等関数
def function_identify(x):
        return x

if __name__ == '__main__':
#結果の出力
        network = init_network()
        x = np.array([1.0, 0.5])
        y = forward(network, x)
        print(y)
```

目次

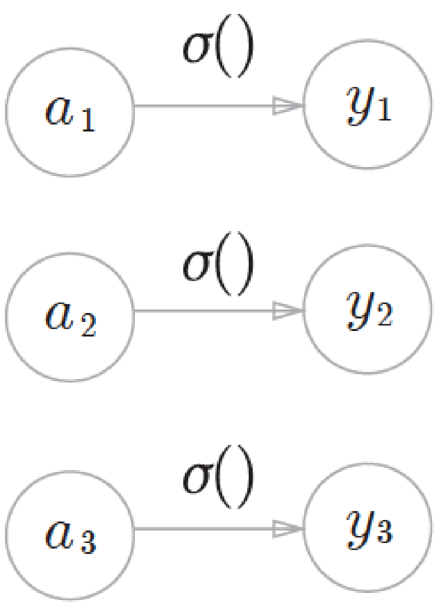
出力層の設計

出力層の活性化関数は分類と回帰によって違う

- 分類(○と✕を当てる問題) -> 恒等関数
- 回帰(値の予測) -> ソフトマックス関数

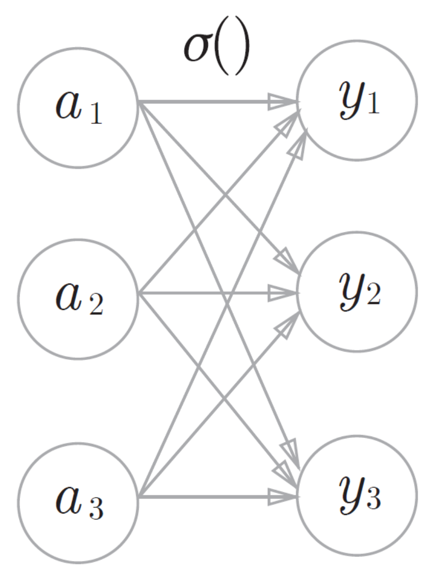
恒等関数

$$y = x$$



ソフトマックス関数

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



ソフトマックス関数の特徴

特徴

- ソフトマックス関数の出力は、0 ~ 1の範囲の値になっており、その総和が 1 になる　-> 確率に対応
- ソフトマックス関数を適応しても、 **A**(x)の各要素の大小関係は変わらない
- ソフトマックス関数は推論フェーズでは省略できる

機械学習の問題を解く手順は「学習」と「推論」に分けられる。最初に学習フェーズでモデル学習を行い、推論フェーズで、学習したモデルを使って未知のデータに対して推論(分類)を行う。

ソフトマックス関数の実装


```
def softmax(a)
    exp_a = np.exp(a)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a
    return y
```

実装上の注意点：指数関数を行うので、オーバーフローになる恐れがあるので、計算結果が不安定になりうる。

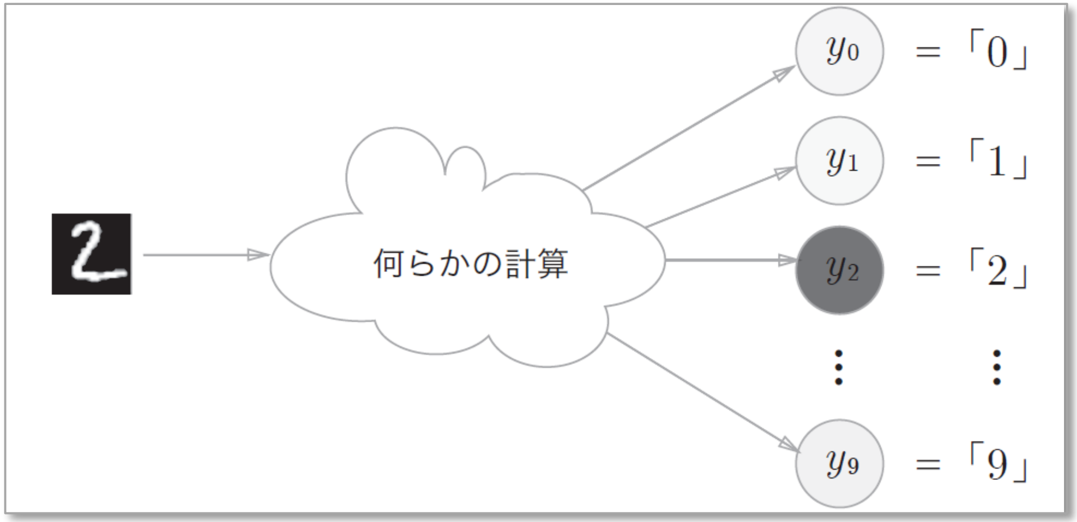
$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n}\exp(a_i)} = \frac{\exp(a_k + C)}{\sum_{i=1}^{n}\exp(a_i + C)}$$

指数関数を行う際には、定数を足し算/引き算しても結果は変わらないので、オーバーフローの対策としては、入力信号の中で最大の値を引くことが一般的

```
def softmax(a)
    c = np.max(a)
    exp_a = np.exp(a - c)
    sum_exp_a = np.sum(exp_a)
    y = exp_a / sum_exp_a
    return y
```

出力層のニューロンの数

出力層のニューロンの数は、解くべき問題に応じて、適宜に決める必要がある。
クラス分類を行う問題では、出力層のニューロンの数は分類したいクラスの数に設定するのが一般的



この例では、y2に該当するクラス、つまり「2」であることを、このニューラルネットワークが予測している

目次

手書き数字認識問題

手書き数字画像の分類問題を行う

- 学習済みのパラメータを使って、ニューラルネットワークの「推論処理」だけを実装していく。この推論処理は、ニューラルネットワークの順方向伝播とも言う。
- 使用するデータセットはMNISTという手書き数字の画像セット（訓練画像が6万枚、テスト画像が1万枚）
- MNISTの画像データは28×28のグレー画像、各ピクセルは0から255までの値を取る

データの読み込み

今回は、mnist.py の 関数 load_mnist()を用いれば、MNISTデータを次のように簡単に読み込める

```
import sys, os
sys.path.append(os.pardir)  # 親ディレクトリのファイルをインポートするための設定
from dataset.mnist import load_mnist

# 最初の呼び出しは数分待ちます…
(x_train, t_train), (x_test, t_test) = \
    load_mnist(flatten=True, normalize=False)

# それぞれのデータの形状を出力
print(x_train.shape) # (60000, 784)
print(t_train.shape) # (60000,)
print(x_test.shape)  # (10000, 784)
print(t_test.shape)  # (10000,)
```

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)」という形式で、読み込んだMNISTデータを返す。
引数として、**load_mnist(normalize=True, flatten=True, one_hot_label=False)**
Normalize：入力画像を 0.0 ~ 1.0の値に正規化するかどうか、Falseすると入力画像は0~255のまま
Flatten：入力画像を1次元配列にするかどうか、Falseのとき、入力画像は1×28×28の3次元配列、Trueのとき、784個の要素からなる1次元配列
one_hot_label：one_hot表現では、正解のラベルが1でそれ以外が0

訓練画像の表示

訓練画像の1枚目を表示してみる

mnist_show.py

```
import sys, os
sys.path.append(os.pardir)
import numpy as np
from dataset.mnist import load_mnist
from PIL import Image

def img_show(img):
    pil_img = Image.fromarray(np.uint8(img))
    pil_img.show()

(x_train, t_train), (x_test, t_test) = \
    load_mnist(flatten=True, normalize=False)

img = x_train[0]
label = t_train[0]
print(label) # 5

print(img.shape)          # (784,)
img = img.reshape(28, 28) # 形状を元の画像サイズに変形
print(img.shape)          # (28, 28)

img_show(img)
```

ニューラルネットワークの推論処理

ネットワークは、入力層を784（画像サイズ：28×28）個、出力層を10個のニューロンで構成する
隠れ層2つ、一つ目の隠れ層は50個、2つ目の層が100個のニューロン
50と100は任意設定

```
def get_data():
    (x_train, t_train), (x_test, t_test) = \
        load_mnist(normalize=True, flatten=True, one_hot_label=False)
    return x_test, t_test

def init_network():
    with open("sample_weight.pkl", 'rb') as f:
        network = pickle.load(f)

    return network

def predict(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 = softmax(a3)

    return y
```

練習１：認識精度計算の実装

予測した答えと正解ラベルとを比較して、正解した割合を認識精度とする

今後は、ニューラルネットワークの構造や学習方法を工夫することで、認識精度を高くする予定

この例では、データを決まった範囲に変換する処理を正規化と言う

入力データに対して、何らかの決まった変換を行うことを前処理という

バッチ処理

	X	W1	W2	W3	→	Y
形状：	100×784	784×50	50×100	100×10		100×10

100枚の画像データのように、まとまりのある入力データをバッチと呼ぶ

バッチ処理によって、1枚当たりの処理時間を大幅に短縮できる

- 計算アルゴリズムが最適化された
- バス帯域の負荷を減らすことができる

練習2 バッチ処理の実装

バッチ処理によって高速に効率良く処理することが可能

練習2 バッチ処理の実装


```
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])
```

バッチ処理によって高速に効率良く処理することが可能

目次

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
- 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
- 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が一般
- 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可能