DeepLeaning

目次

[準備 3](#_Toc515213537)

[パーセプトロン 3](#_Toc515213538)

[ニューラルネットワーク 5](#_Toc515213539)

[活性化関数 5](#_Toc515213540)

[3層ニューラルネットワークの実装 7](#_Toc515213541)

[手書き数字認識 7](#_Toc515213542)

[ニューラルネットワークの学習 8](#_Toc515213543)

[損失関数 8](#_Toc515213544)

[バッチ学習 9](#_Toc515213545)

[誤差伝播法 10](#_Toc515213546)

[計算グラフの逆伝播 10](#_Toc515213547)

[Python予備知識 12](#_Toc515213548)

[Pickle 12](#_Toc515213549)

# 準備

* 書籍：ゼロから作るDeepLearning　Pythonで学ぶディープラーニングの理論と実装
* ソース：<https://github.com/oreilly-japan/deep-learning-from-scratch>
* 使用するライブラリ
  + numpy
  + matplotlib

# パーセプトロン

複数の信号を入力として受け取り、一つの信号（0 or 1）を出力する

­

ポイント！

* 入力された信号に対して固有の重みを持たせる
* 適切なバイアス値を設定し、0を閾値として発火有無を制御する

▽AND,OR,NAND,XORの実装

def **AND**(x1, x2):

ans = 0

x = np.array([x1, x2, 1])

w = 1, 1, -1

y = np.dot(x, w)

if(y <= 0):

ans = 0

else:

ans = 1

return ans

def **OR**(x1, x2):

ans = 0

x = np.array([x1, x2, 1])

w = 1, 1, 0

y = np.dot(x, w)

if(y <= 0):

ans = 0

else:

ans = 1

return ans

def **NAND**(x1, x2):

ans = 0

x = np.array([x1, x2, 1])

w = -1, -1, 1.1

y = np.dot(x, w)

if(y <= 0):

ans = 0

else:

ans = 1

return ans

def **XOR**(x1, x2):

# 第一層

s1 = OR(x1, x2)

s2 = NAND(x1, x2)

# 第二層

return AND(s1,s2)

# ニューラルネットワーク

## 活性化関数

下記図でいうh()のこと

入力信号に重み付けをした結果に対して、出力信号に変換する関数

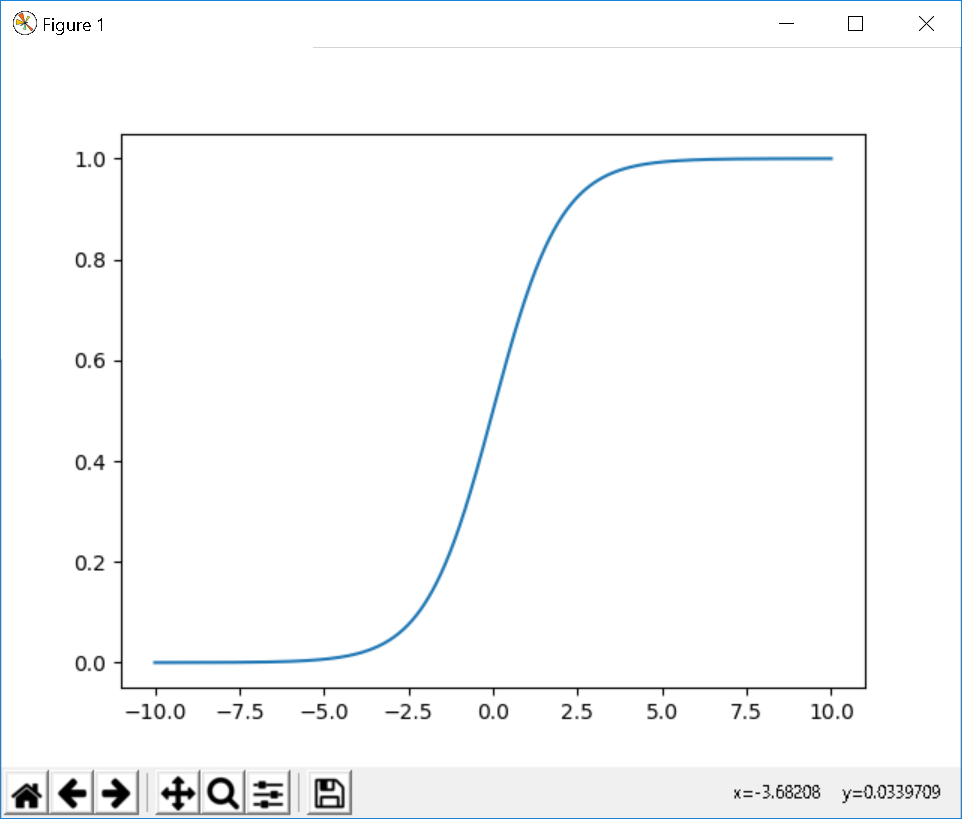
h()

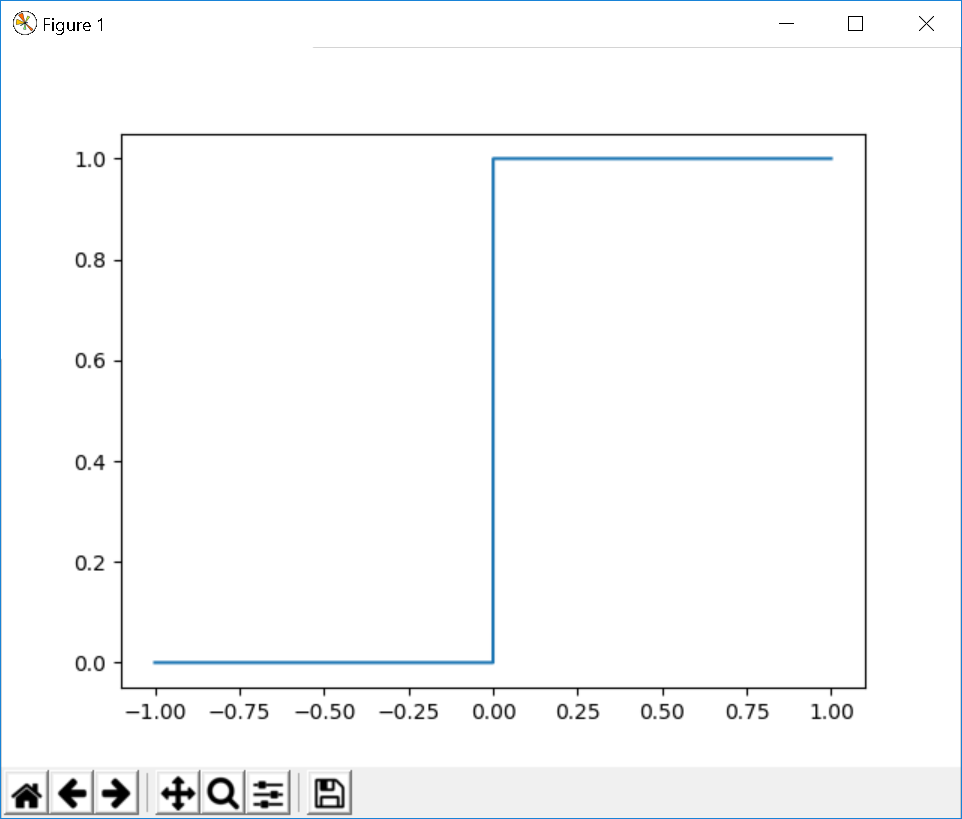
w1

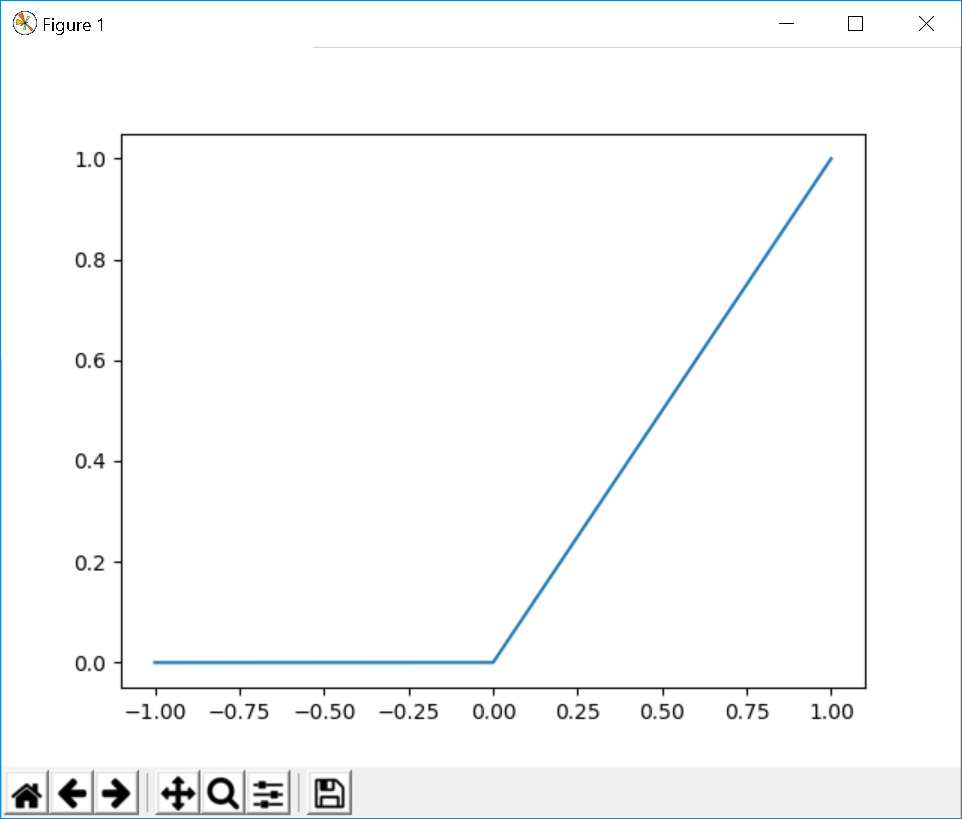
w2

b

* シグモイド関数



* ステップ関数
  + 
* ReLU関数



▽実装例

def **ReLU**(x):

return np.max(0, x)

* ソフトマックス関数
  + 1段目の式が基本形
  + １段目だと大きい値を持った要素があった場合に∞／∞の計算になりうる（プログラム的に）
  + ３段目のC’に要素のうち最大のものを当てはめることで回避する

▽実装例

def **softmax**(x):

# a / b

max = np.max(x)

a = np.array([np.exp(xi) - max for xi in x])

b = np.sum(a)

return a / b;

## 3層ニューラルネットワークの実装

XORを3層ニューラルネットワークで実装する

▽実装

def **XOR2**(x1, x2):

x = np.array([x1, x2, 1])

w1 = np.array([[1, 1, 0],[-1, -1, 1.1], [0, 0, 1]])

w2 = np.array([1, 1, -1])

a1 = np.array([step(a) for a in np.dot(w1, x.T)])

a2 = step(np.dot(w2, a1.T))

return a2

## 手書き数字認識

※準備中

# ニューラルネットワークの学習

ニューラルネットワークの学習では訓練データとテストデータを使用する

* 訓練データ
  + 学習を行い、パラメータの更新を行う
* テストデータ
  + 学習を行った結果を評価する
  + 学習データと別のテストデータを用意することで過学習の状態を避ける

## 損失関数

●平均二乗誤差

●交差エントロピー誤差

※δは微小量

ポイント！

* + １段目の式が基本形
  + １段目だとの場合にEが発散
  + そのため２段目で微小量を加える
  + は正解ラベルのみ１でその他は０

## バッチ学習

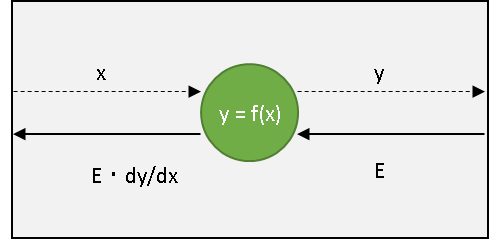
訓練データに対する損失関数を求め、損失関数が小さくなるようにパラメータを更新すること

●ミニバッチ学習

* 大量の訓練データすべてを対象に損失関数を求めるのは時間がかかる
* そのためランダムで訓練データの一部を選出し、選ばれたデータのみで損失関数を求める

# 誤差伝播法

## 計算グラフの逆伝播



* 数値微分はシンプルかつ実装が楽（統一的）
* 誤差伝播法は、微分後の関数をプログラムに教えてあげることで計算効率を向上させる

レイヤ実装のポイント！

* + 入出力を統一する

⇒コレクションでレイヤを定義することでイテレータを使用できる

* + forwardは入力信号を受け取り、出力信号を返す
  + backwardは出力信号を受け取り、入力信号を返す
  + backwardでその他のパラメータ等の微分値が欲しい場合は、フィールド変数に保持しておく

## ReLUレイヤの実装

▽実装例

class **ReluLayer**:

def **\_\_init\_\_**(*self*):

*self*.mask = None

def **forward**(*self*, x, y):

*self*.mask = (x <= 0)

dout = x.copy()

dout[*self*.mask] = 0

return dout

def **backward**(*self*, dout):

dx = dout.copy()

dx[*self*.mask] = 0

return dx

ポイント！

* + 入力値が何だったかによって逆伝播を行う必要がある
  + 入力値を保存しておく必要がある（実装例ではマスクする場所を保存している）

## Sigmoidレイヤ

▽微分

▽実装例

class **SigmoidLayer**:

def **\_\_init\_\_**(*self*):

*self*.out = None

def **forward**(*self*, x):

out = 1 / ( 1 + np.exp(-x) )

*self*.out = out

return out

def **backward**(*self*, dout):

dx = dout \* (1.0 - *self*.out) \* *self*.out

return dx

ポイント！

* + フォワードの出力値を保存しておくこと

## AFFINEレイヤ

▽実装例

class **Affine**:

def **\_\_init\_\_**(*self*, w, b):

*self*.w = w

*self*.b = b

*self*.x = None

*self*.dw = None

*self*.db = None

def **forward**(*self*, x):

*self*.x = x

out = np.dot(*self*.x, *self*.w) + *self*.b

return out

def **backward**(*self*, dout):

dx = np.dot(dout, *self*.w.T)

*self*.dw = np.dot(*self*.x.T, dout)

*self*.db = np.sum(dout, axis=0)

return dx

## ソフトマックス＆交差エントロピー誤差レイヤ

▽実装例

class **SoftmaxWithLoss**:

def **\_\_init\_\_**(*self*):

*self*.loss = None

*self*.y = None

*self*.t = None

def **forward**(*self*, x, t):

*self*.t = t

*self*.y = softmax(x)

*self*.loss = cross\_entropy\_error(*self*.y, *self*.t)

return *self*.loss

def **backward**(*self*, dout=1):

batch\_size = *self*.t.shape[0]

if *self*.t.size == *self*.y.size:

dx = (*self*.y - *self*.t) / batch\_size

else:

dx = *self*.y.copy()

dx[np.arange(batch\_size), *self*.t] -= -1

dx = dx / batch\_size

return dx

# Python予備知識

## Pickle

プログラム実行中のオブジェクトをファイルとして保存する機能

⇒これを利用することで2回目以降の呼び出しが高速に行われる！

▽実装例（ファイルへの書込み）

import pickle

with open(save\_file, *'wb'*) as f:

pickle.dump(dataset, f, -1)

▽実装例（ファイルからの読込み）

import pickle

with open(save\_file, *'rb'*) as f:

dataset = pickle.load(f)

※datasetが保存対象のオブジェクト