DeepLeaning

目次

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

$$y = \begin{cases} 0 & (b + \omega_1 x_1 + \omega_2 x_2 \leq 0) \\ 1 & (b + \omega_1 x_1 + \omega_2 x_2 > 0) \end{cases}$$

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):
```

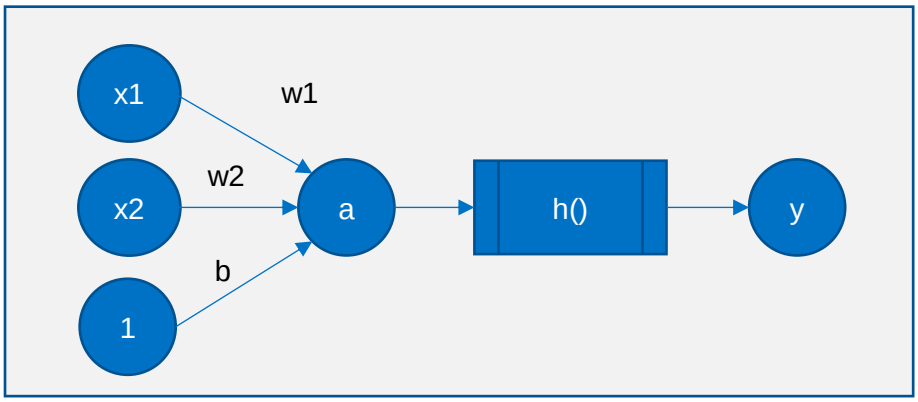
```
    ans = 0
    x = np.array([x1, x2, 1])
    w = -1, -1, 1.1
    y = np.dot(x, w)
    if(y <= 0):
        ans = 0
    else:
        ans = 1
    return ans

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

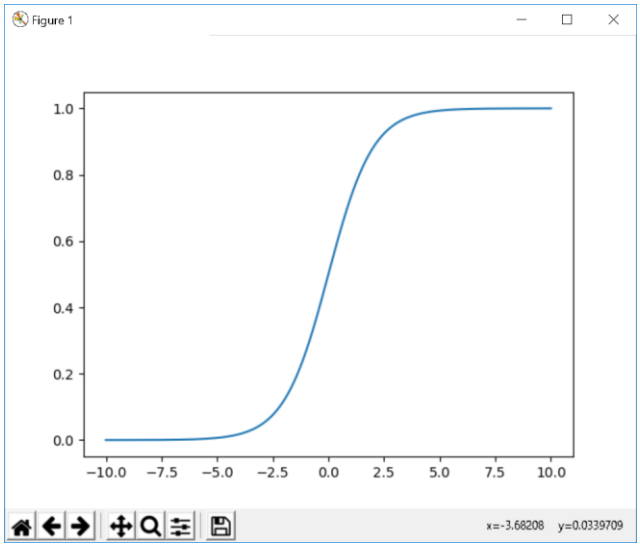
活性化関数

下記図でいうh()のこと

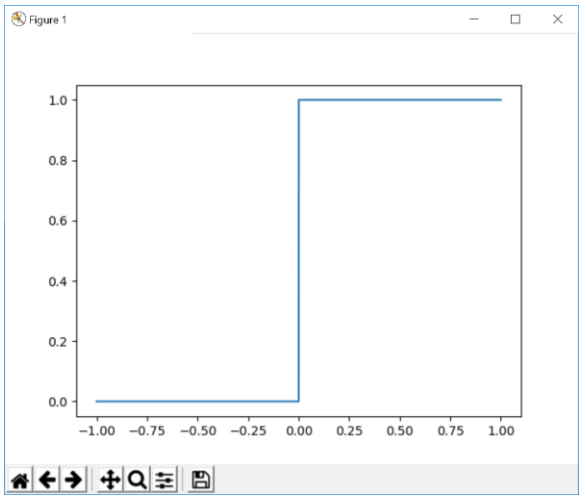
入力信号に重み付けをした結果に対して、出力信号に変換する関数



- シグモイド関数
  - $h(x) = \frac{1}{1+\exp(-x)}$

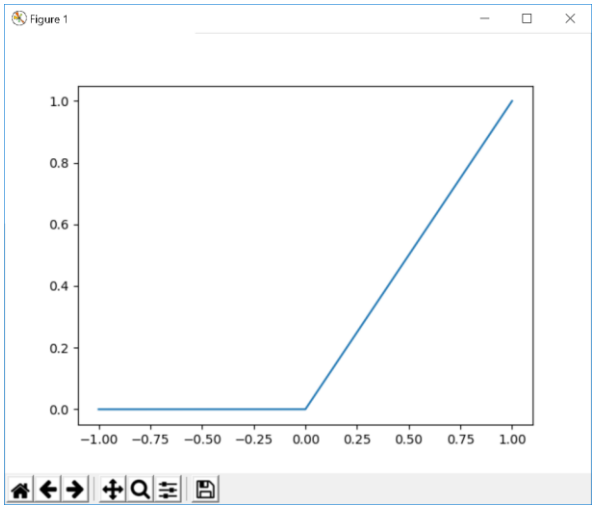


- ステップ関数



- ReLU 関数

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



▽実装例

```
def ReLU(x):
    return np.max(0, x)
```

- ソフトマックス関数

$$
\begin{aligned}
y_i &= \frac{\exp(a_i)}{\sum_n \exp(a_n)} \\
&= \frac{C \exp(a_i)}{C \sum_i \exp(a_n)} \\
&= \frac{\exp(a_i - C')}{\sum_n \exp(a_n - C')}
\end{aligned}
$$

  - 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

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)^2$$

●交差エントロピー誤差

※δは微小量

$$\mathrm{E} = -\sum_{\mathrm{k}} t_k \log y_k$$
$$= -\sum_{\mathrm{k}} t_k \log(y_k + \delta)$$

ポイント！

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

バッチ学習

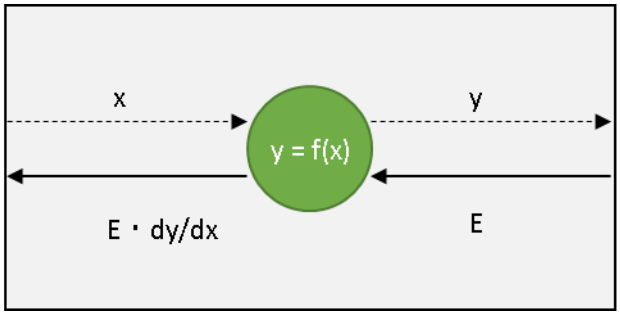
訓練データに対する損失関数を求め、損失関数が小さくなるようにパラメータを更新すること

●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

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

$$\frac{\partial y}{\partial x} = \frac{\partial}{\partial x}\frac{1}{1+exp(-x)}$$
$$= \frac{-exp(-x)}{(1+exp(-x))^2}$$
$$= \frac{1}{1+exp(-x)}\left(1-\frac{1}{1+exp(-x)}\right)$$
$$\therefore \frac{\partial y}{\partial x} = y(1-y)$$

▽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

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