DAY1

ニューラルネットワーク全体像

入力層〇 ⇒w(1)b(1) 〇 ⇒w(2)b(2) 〇出力層
中間層

確認テスト

ディープラーニングは、結局何をやろうとしているか 2 行以内で述べよ。また、次の中のどの値の最適化が最終目的か。

全て選べ。

A.ニューラルネットワークを活用し、答えを学習することで、入力値から出力値（答え）を得ることができる。

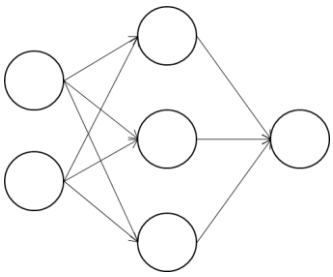
①入力値[X] ②出力値[Y]③重み[W]④バイアス[b]⑤総入力[u] ⑥中間層入力[z]⑦学習率[ρ]

次のネットワークを紙にかけ。

入力層：2 ノード 1 層

中間層： 3ノード 2 層

出力層：1 ノード 1 層



ニューラルネットワークでできること

・回帰（結果予測、ランキング等）

連続する実数値を取る関数の近似

・分類（写真判定、手書き文字認識等）

性別（男あるいは女）や動物の種類など離散的な結果を予想するための分析

Section1：入力層～中間層

◇要点

入力層

人工ニューロンが最初に情報を受け取るのが入力層。手書きで「0」と書かれた画像を人工ニューロンに認識させる場合、入力層では画像の 1 ピクセルを入力値として受け取る。入力層で受け取った情報は、ニューロン同士の結合の強度に応じて、優先順位が決定される仕組み。

中間層

入力層から情報を受け継ぎ、さまざまな計算を行うのが中間層です。中間層が多いほど複雑な分析ができ、中間層が 3 層以上あるニューラルネットワークをディープラーニングと呼ぶ。中間層の数に決まりはなく、扱う情報にあわせた任意での設定が可能。

◇実装演習結果キャプチャー又はサマリーと考察

```
# 順伝播（3 層・複数ユニット）

# ウェイトとバイアスを設定
# ネートワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")
    network = {}

    #試してみよう
    #_各パラメータの shape を表示
    #_ネットワークの初期値ランダム生成

    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
    network['W2'] = np.array([
        [0.1, 0.4],
        [0.2, 0.5],
        [0.3, 0.6]
    ])
    network['W3'] = np.array([
        [0.1, 0.3],
        [0.2, 0.4]
    ])
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2])
    network['b3'] = np.array([1, 2])

    print_vec("重み 1", network['W1'] )
    print_vec("重み 2", network['W2'] )
    print_vec("重み 3", network['W3'] )
    print_vec("バイアス 1", network['b1'] )
    print_vec("バイアス 2", network['b2'] )
    print_vec("バイアス 3", network['b3'] )

    return network

# プロセスを作成
# x：入力値
```

```
def forward(network, x):

    print("##### 順伝播開始 #####")

    W1, W2, W3 = network['W1'], network['W2'], network['W3']
    b1, b2, b3 = network['b1'], network['b2'], network['b3']

    # 1 層の総入力
    u1 = np.dot(x, W1) + b1

    # 1 層の総出力
    z1 = functions.relu(u1)

    # 2 層の総入力
    u2 = np.dot(z1, W2) + b2

    # 2 層の総出力
    z2 = functions.relu(u2)

    # 出力層の総入力
    u3 = np.dot(z2, W3) + b3

    # 出力層の総出力
    y = u3

    print_vec("総入力 1", u1)
    print_vec("中間層出力 1", z1)
    print_vec("総入力 2", u2)
    print_vec("出力 1", z1)
    print("出力合計: " + str(np.sum(z1)))

    return y, z1, z2

# 入力値
x = np.array([1., 2.])
print_vec("入力", x)

# ネットワークの初期化
network =  init_network()

y, z1, z2 = forward(network, x)
```

ソースコード「1_1_forward_propagation」の順伝播（単層・単ユニット）において、重みおよびバイアスを乱数で初期化

```
W = np.zeros(2)                          # W の各値に「0」を代入
W = np.ones(2)                           # W の各値に「1」を代入
W = np.random.rand(2)                    # W の各値に 0~1 の間の乱数を代入
W = np.random.randint(5, size=(2))   # W の各値に 0~4 の間の整数の乱数を代入

b = np.random.rand()                     # b の値に 0~1 の間の乱数を代入
b = np.random.rand() * 10 -5             # b の値に-5~5 の間の乱数を代入
```

・動物分類の実例を入れる

猫　ひげの数、耳の大きさ、体重、手の数、足の数、しっぽの有無・・・

・数式を Python で記載　数式：u = w1x1+w2x2＋・・・＋b = Wx+b

```
u1 = np.dot(x, W1) + b1
```

・中間層の出力を定義しているソース

```
z = functions.relu(u)
z = functions.sigmoid(u)
```

Section2：活性化関数

◇要点

確認（復習）テスト線形と非線形の違いを図にかいて簡易に説明せよ。

入力値の値によって、次の層への信号の ON/OFF や強弱を定める働きをもつ。

中間層用の活性化関数

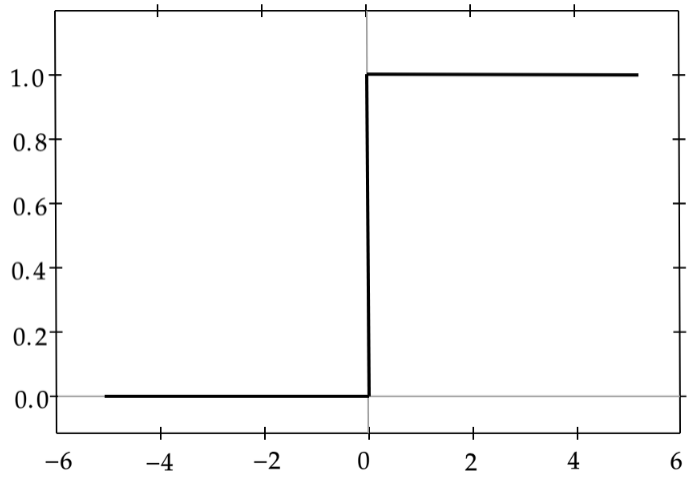
・ReLU 関数

・シグモイド（ロジスティック）関数

・ステップ関数

活性化関数：ステップ関数

しきい値を超えたら発火する関数であり、出力は常に 1 か 0 。パーセプトロン（ニューラルネットワークの前身）で利用された関数。

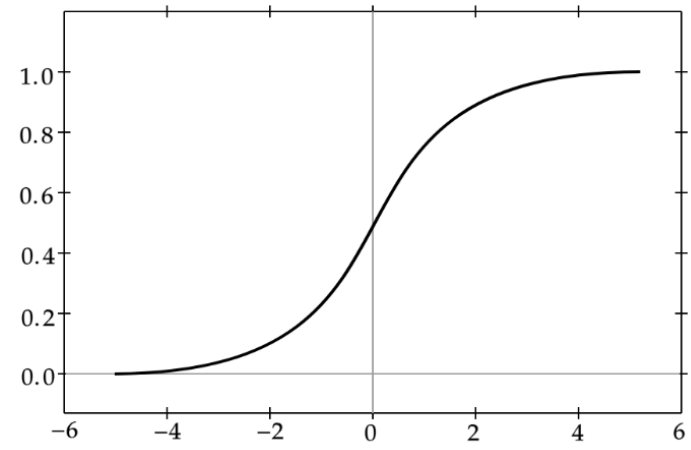


活性化関数：シグモイド関数

0 ～ 1 の間を緩やかに変化する関数で、ステップ関数では ON/OFF しかない状態に対し、信号の強弱を伝えられるようになり、予想ニューラルネットワーク普及のきっかけとなった。

数式

$$f(u) = \frac{1}{1 + e^{-u}}$$

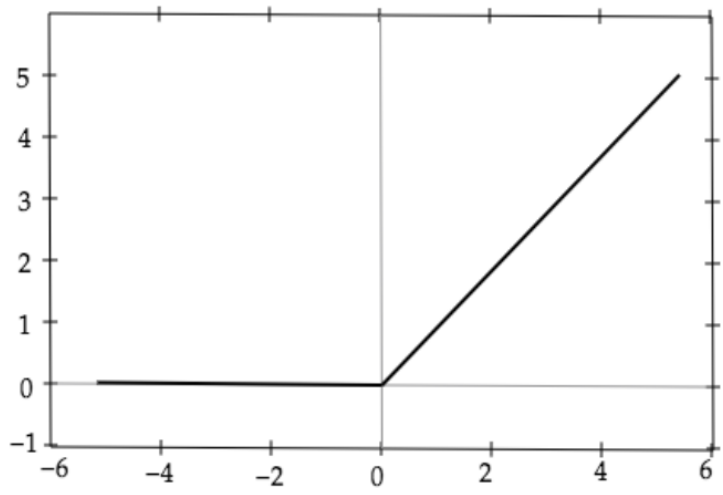


活性化関数：RELU 関数

今最も使われている活性化関数勾配消失問題の回避とスパース化に貢献することで良い成果をもたらしている。

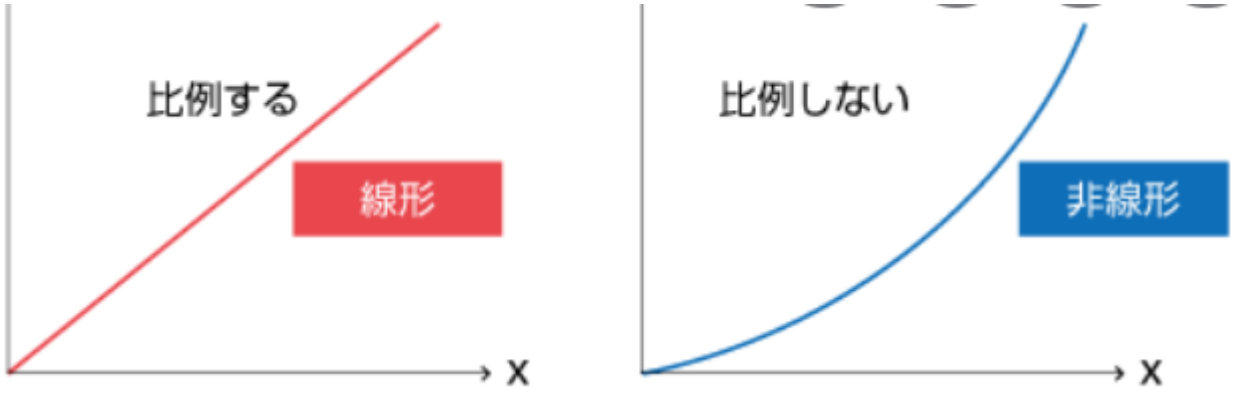
数式

$$\mathrm{f}(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leqq 0) \end{cases}$$



◇実装演習結果キャプチャー又はサマリーと考察

・線形と非線形の違いを図にかいて簡易に説明



Section3:出力層

◇要点

入力層と中間層で重みをかけ、活性化関数で処理された値が示されるのが出力層

誤差関数

出力層の出力結果とあらかじめ用意された正解値を比較し、その誤差を表現する関数。深層学習では、誤差をより小さくするために各入力層の重みを更新することで、モデルの精度を上げる。誤差関数は、平均 2 乗誤差や交差エントロピーを用いる

活性化関数

出力層では中間層と異なる活性化関数が使用される

出力層用の活性化関数

・ソフトマックス関数

・恒等写像

・シグモイド関数（ロジスティック関数）

◇実装演習結果キャプチャー又はサマリーと考察

$$E_n(\mathbf{w}) = \frac{1}{2}\sum_{j=1}^{J}(y_j - d_j)^2 = \frac{1}{2}\|(\mathbf{y}-\mathbf{d})\|^2$$

・なぜ、引き算でなく二乗するのか。
　±ゼロにならないようにするため。
・1/2 の意味
　計算をしやすくするためで特に意味はない。

Section4:勾配降下法

◇要点
深層学習の目的は、学習を通して誤差を最小にするネットワークを作成すること
⇒誤差 E(w)を最小化するパラメータ w を発見すること
勾配降下法を用いてパラメータを最適化する

・勾配降下法

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E$$

該当するソースコード

```
network[key] -= learning_rate * grad[key]
```

$$\nabla E = \frac{\partial E}{\partial \mathbf{w}} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M}\right]$$

該当するソースコード

```
grad = backward(x, d, z1, y)
```

学習率ε
　学習率が大きすぎた場合、最小値にいつまでもたどり着かず発散する。
　学習率が小さすぎた場合、発散することはないが、収束するまでに時間がかかる。
勾配降下法の種類
　Momentum/AdaGrad/Adadelta/Adam/RMSProp

・確率的勾配降下法

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_n$$

　ランダムに抽出したサンプルの誤差

確率的勾配降下法のメリット
・データが冗長な場合の計算コストの軽減
・望まない局所極小解に収束するリスクの軽減
・オンライン学習ができる

・ミニバッチ勾配降下法

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_t$$

$$E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n$$

$$N_t = |D_t|$$

確率的勾配降下法のメリットを損なわず、計算機の計算資源を有効利用できる
→CPU を利用したスレッド並列化や GPU を利用した SIMD 並列化

◇実装演習結果キャプチャー又はサマリーと考察

・オンライン学習とは何か。
A.学習データが入ってくるたびに都度学習パラーメータを更新し、学習を進めていく方法。
・以下の式の意味を図に書いて説明

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_t$$

W(t)⇒[-εΔEt]⇒W(t+1) ⇒[-εΔEt+1]⇒W(t+2)

Section5：誤差逆伝播法

◇要点

$$\nabla E = \frac{\partial E}{\partial \mathbf{w}} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M} \right] \quad \frac{\partial E}{\partial w_m} \approx \frac{E(w_m + h) - E(w_m - h)}{2h}$$

数値微分
プログラムで微小な数値を生成し擬似的に微分を計算する一般的な手法
各パラメータ Wmそれぞれについて E(Wm+h)や E(Wm-h)を計算するために、順伝播の計算を繰り返し行う必要があり負担が大きいため、誤差逆伝播法を利用する。
誤差逆伝播法
算出された誤差を、出力層側から順に微分し、前の層前の層へと伝播。最小限の計算で各パラメータでの微分値を解析的に計算する手法
計算結果（=誤差）から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる

◇実装演習結果キャプチャー又はサマリーと考察
∂E/∂y
delta2 = functions.d_mean_squared_error(d, y)
∂E/∂y ∂y/∂u
delta2 = functions.d_mean_squared_error(d, y)
∂E/∂y ∂y/∂u ∂u/∂w(2)ji

grad['W2'] = np.dot(z1.T, delta2)

・誤差逆伝播法では不要な再帰的処理を避ける事が出来る。既に行った計算結果を保持しているソースコードを抽出せよ。

```
#  出力層でのデルタ
 delta2 = functions.d_sigmoid_with_loss(d, y)
 #  中間層でのデルタ
 delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
```

1_1_forward_propagation_after.ipynb

準備

Googleドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')
```

sys.pathの設定

以下では，Googleドライブのマイドライブ直下にDNN_codeフォルダを置くことを仮定しています．必要に応じて，パスを変更してください。

```
import sys
```

```
import numpy as np
from common import functions

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
    print("shape: " + str(vec.shape))
    print("")
```

順伝播（単層・単ユニット）

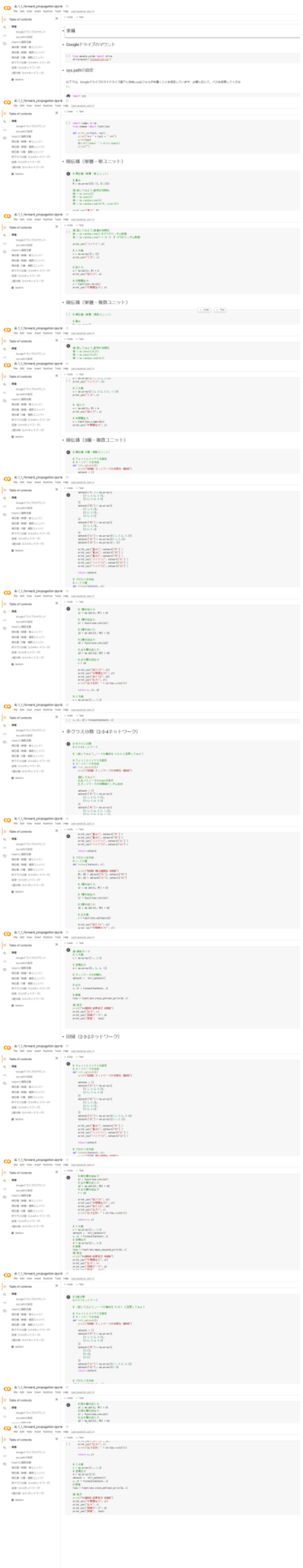
順伝播（単層・複数ユニット）

順伝播（3層・複数ユニット）

多クラス分類（2-3-4ネットワーク）

回帰（2-3-2ネットワーク）

2値分類（2-3-1ネットワーク）


1_1_forward_propagation.ipynb
Table of contents
準備
Googleドライブのマウント
sys.pathの設定
順伝播（単層・単ユニット）
順伝播（単層・複数ユニット）
順伝播（3層・複数ユニット）
多クラス分類（2-3-4ネットワーク）
回帰（2-3-2ネットワーク）

1_2_back_propagation.ipynb

File Edit View Insert Runtime Tools Help Last edited on Aug 30, 2019

Table of contents

- 準備
  - Googleドライブのマウント
  - sys.pathの設定
- importと関数定義
- メインプログラム
- Section

+ Code + Text

準備

Googleドライブのマウント

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

sys.pathの設定

以下では，Googleドライブのマイドライブ直下にDNN_codeフォルダを置くことを仮定しています．必要に応じて，パスを変更してください．

```
[ ] import sys
```

1_2_back_propagation.ipynb

File Edit View Insert Runtime Tools Help Last edited on Aug 30, 2019

Table of contents

- 準備
  - Googleドライブのマウント
  - sys.pathの設定
- importと関数定義
- メインプログラム
- Section

+ Code + Text

```
[ ] import numpy as np
    from common import functions
    import matplotlib.pyplot as plt

    def print_vec(text, vec):
        print("*** " + text + " ***")
        print(vec)
        #print("shape: " + str(x.shape))
        print("")
```

メインプログラム

```
# ウェイトとバイアスを設定
# ネットワークを作成
def init_network():
    print("###### ネットワークの初期化 ######")

    network = {}
    network['W1'] = np.array([
        [0.1, 0.3, 0.5],
        [0.2, 0.4, 0.6]
    ])
```

1_2_back_propagation.ipynb

File Edit View Insert Runtime Tools Help Last edited on Aug 30, 2019

Table of contents

- 準備
  - Googleドライブのマウント
  - sys.pathの設定
- importと関数定義
- メインプログラム
- Section

+ Code + Text

```
    network['b1'] = np.array([0.1, 0.2, 0.3])
    network['b2'] = np.array([0.1, 0.2])

    print_vec("重み1", network['W1'])
    print_vec("重み2", network['W2'])
    print_vec("バイアス1", network['b1'])
    print_vec("バイアス2", network['b2'])

    return network

# 順伝播
def forward(network, x):
    print("###### 順伝播開始 ######")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    u1 = np.dot(x, W1) + b1
    z1 = functions.relu(u1)
    u2 = np.dot(z1, W2) + b2
    y = functions.softmax(u2)

    print_vec("総入力1", u1)
    print_vec("中間層出力1", z1)
    print_vec("総入力2", u2)
    print_vec("出力1", y)
    print("出力合計: " + str(np.sum(y)))
```

1_2_back_propagation.ipynb

File Edit View Insert Runtime Tools Help Last edited on Aug 30, 2019

Table of contents

- 準備
  - Googleドライブのマウント
  - sys.pathの設定
- importと関数定義
- メインプログラム
- Section

+ Code + Text

```
    grad = {}

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    #  出力層でのデルタ
    delta2 = functions.d_sigmoid_with_loss(d, y)
    #  b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    #  W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    #  中間層でのデルタ
    delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    #  W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    print_vec("偏微分_dE/du2", delta2)
    print_vec("偏微分_dE/du2", delta1)

    print_vec("偏微分_重み1", grad["W1"])
    print_vec("偏微分_重み2", grad["W2"])
    print_vec("偏微分_バイアス1", grad["b1"])
    print_vec("偏微分_バイアス2", grad["b2"])
```

1_2_back_propagation.ipynb

File Edit View Insert Runtime Tools Help Last edited on Aug 30, 2019

Table of contents

- 準備
  - Googleドライブのマウント
  - sys.pathの設定
- importと関数定義
- メインプログラム
- Section

+ Code + Text

```
# 学習率
learning_rate = 0.01
network =  init_network()
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)

grad = backward(x, d, z1, y)
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key]  -= learning_rate * grad[key]

print("###### 結果表示 ######")


print("###### 更新後パラメータ ######")
print_vec("重み1", network['W1'])
print_vec("重み2", network['W2'])
print_vec("バイアス1", network['b1'])
print_vec("バイアス2", network['b2'])
```