④勾配降下法_実装演習

=====

目次:

1.確率勾配降下法

2.計算時に用いる活性化関数の導関数の実装

3.数値微分の実装

【要約】

・計算出力と正解データの比較による誤差を利用して、パラメータ[誤差・バイアス]の更新を行う

・w = w -ε*▽E

・確率的勾配降下法;ランダムに抽出したサンプルデータの誤差を用い、学習を進める方法

・中間層の誤差計算には、誤差逆伝播法を利用(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)

・誤差計算時には、微分計算が必要になるが、微分式の実装は解析的計算で実装(数値計算[数値微分]で実装すると、計算量が多くなってしまう)

In [3]:

```
import sys
sys.path.append('C:/Users/NIF/Desktop/(削除)rabitt/DNN_code_colab_lesson_1_2/DNN_code_colab_lesson_1_2')

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("")
```

=====

1.確率勾配降下法

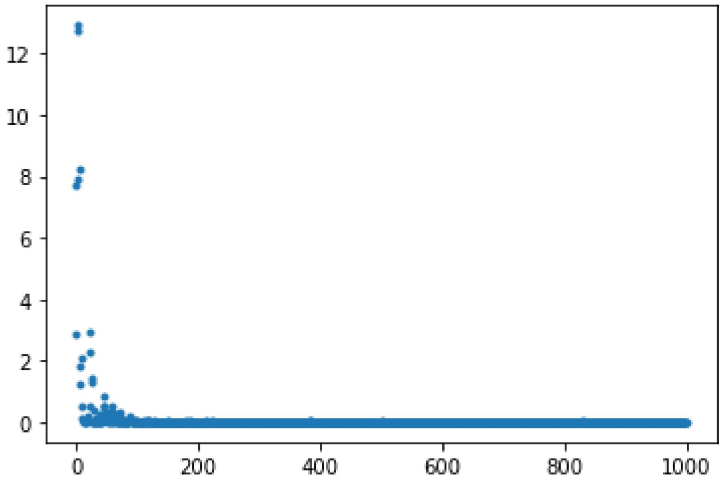
In [4]:

```
# サンプルとする関数
#yの値を予想するAI

def f(x):
    y = 3 * x[0] + 2 * x[1]
    return y

# 初期設定
def init_network():
    # print("##### ネットワークの初期化 #####")
    network = {}
    nodesNum = 10
    network['W1'] = np.random.randn(2, nodesNum)
    network['W2'] = np.random.randn(nodesNum)
    network['b1'] = np.random.randn(nodesNum)
    network['b2'] = np.random.randn()

    # 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)

    ## 試してみよう
    #z1 = functions.sigmoid(u1)

    u2 = np.dot(z1, W2) + b2
    y = 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)))

    return z1, y

# 誤差逆伝播
def backward(x, d, z1, y):
    # print("¥n##### 誤差逆伝播開始 #####")

    grad = {}

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

    # 出力層でのデルタ
    delta2 = functions.d_mean_squared_error(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)

    ## 試してみよう
    delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

    delta1 = delta1[np.newaxis, :]
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    x = x[np.newaxis, :]
    # W1の勾配
    grad['W1'] = np.dot(x.T, 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"])

    return grad

# サンプルデータを作成
data_sets_size = 100000
data_sets = [0 for i in range(data_sets_size)]

for i in range(data_sets_size):
    data_sets[i] = {}
    # ランダムな値を設定
    data_sets[i]['x'] = np.random.rand(2)

    ## 試してみよう_入力値の設定
    # data_sets[i]['x'] = np.random.rand(2) * 10 -5 # -5〜5のランダム数値

    # 目標出力を設定
    data_sets[i]['d'] = f(data_sets[i]['x'])

losses = []
# 学習率
learning_rate = 0.07

# 抽出数
epoch = 1000

# パラメータの初期化
network = init_network()
# データのランダム抽出
random_datasets = np.random.choice(data_sets, epoch)

# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key]  -= learning_rate * grad[key]

    # 誤差
    loss = functions.mean_squared_error(d, y)
```

```
    losses.append(loss)

print("##### 結果表示 #####")
lists = range(epoch)


plt.plot(lists, losses, '.')
# グラフの表示
plt.show()
```

結果表示



→出力層:2乗和誤差を持ちて、▽Eを計算

→中間層:出力層の▽E及び中間層の活性化関数(シグモイド関数)を用いて、▽Eを計算[誤差逆伝播]

→勾配降下法:w = w - ε*▽Eにより重み及びバイアスのパラメータ値を更新

→エポック数を1000回と設定し、学習を重ねることで、出力の誤差を下げている

→確率的勾配降下法を採用しているため、ランダムにデータを選び学習を行っている

=====

2.計算時に用いる活性化関数の導関数の実装

In [5]:

```
# 活性化関数の導関数
# シグモイド関数（ロジスティック関数）の導関数
def d_sigmoid(x):
    dx = (1.0 - sigmoid(x)) * sigmoid(x)
    return dx

# ReLU関数の導関数
def d_relu(x):
    return np.where( x > 0, 1, 0)

# ステップ関数の導関数
def d_step_function(x):
    return 0

# 平均二乗誤差の導関数
def d_mean_squared_error(d, y):
    if type(d) == np.ndarray:
        batch_size = d.shape[0]
        dx = (y - d)/batch_size
    else:
        dx = y - d
    return dx
```

→基本的に数式を解析的に微分した結果のとおりに実装

=====

3.数値微分の実装

In [7]:

```
# 数値微分
def numerical_gradient(f, x):
    h = 1e-4
    grad = np.zeros_like(x)

    for idx in range(x.size):
        tmp_val = x[idx]
        # f(x + h)の計算
        x[idx] = tmp_val + h
        fxh1 = f(x)

        # f(x - h)の計算
        x[idx] = tmp_val - h
        fxh2 = f(x)

        grad[idx] = (fxh1 - fxh2) / (2 * h)
        # 値を元に戻す
        x[idx] = tmp_val

    return grad
```

→f(x+h) - f(x-h)/ 2h の式のとおり

→中心差分で計算(前方差分などの場合は、誤差が生じる)

→数値微分で実装するデメリットとして、計算量が多くなってしまうことが挙げられる

In [ ]: