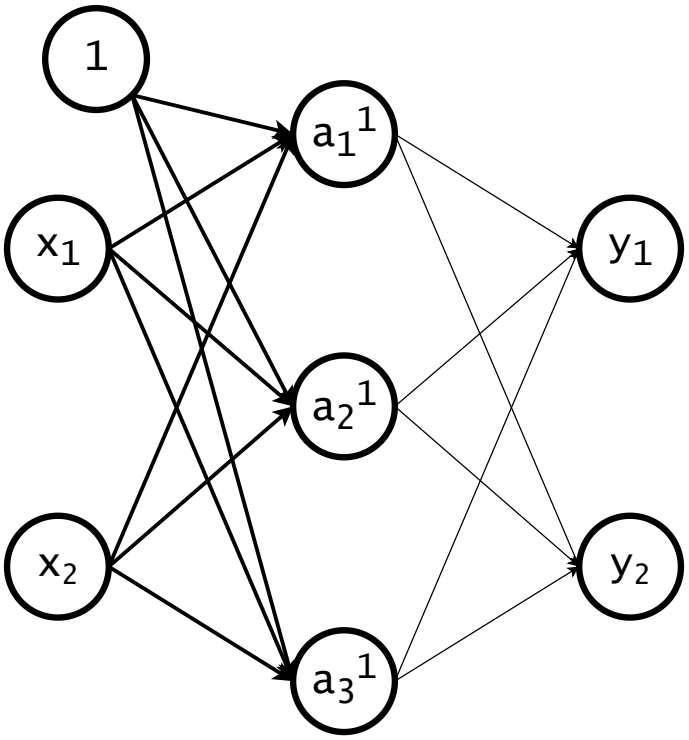
深層学習 DAY1　レポート

2024 年 06 月 10 日 新規作成

Section1:入力層~中間層



ニューラルネットワークを図で表すと上記のようになる。左から入力層、中間層、出力層となっており、信号は入力層から出力層へ順に伝わる。ニューラルネットワークはある種の信号変換装置である。図中の丸はノードやニューロンと呼ばれ、矢印の方向に信号が伝達される。各矢印には重みがあり、信号に重みを掛けた値が次のノードに伝わる。矢印の重みとは言わば信号の伝わりやすさで、重みが大きいほどその矢印の元のノードの影響を強く受ける。中間層のノードには、入力層からの信号にそれぞれ重みを掛けた値が集まる。さらに、各ノードにはバイアスという定数項が加えられる。これらの合計値は活性化関数を通過し、その結果が次の層に伝達される。

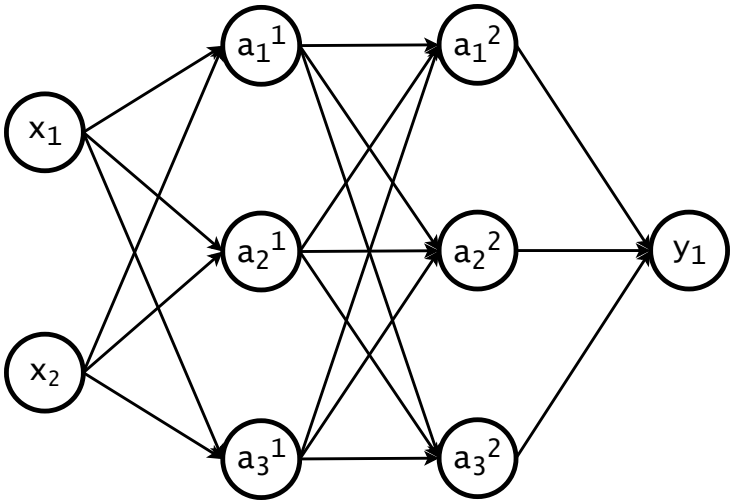
$$
\begin{aligned}
a_1^1\text{の入力値} &= \omega_{11}^1 x_1 + \omega_{21}^1 x_2 + b_1^1 \\
a_2^1\text{の入力値} &= \omega_{12}^1 x_1 + \omega_{22}^1 x_2 + b_2^1 \\
a_3^1\text{の入力値} &= \omega_{13}^1 x_1 + \omega_{23}^1 x_2 + b_3^1
\end{aligned}
$$

確認テスト１

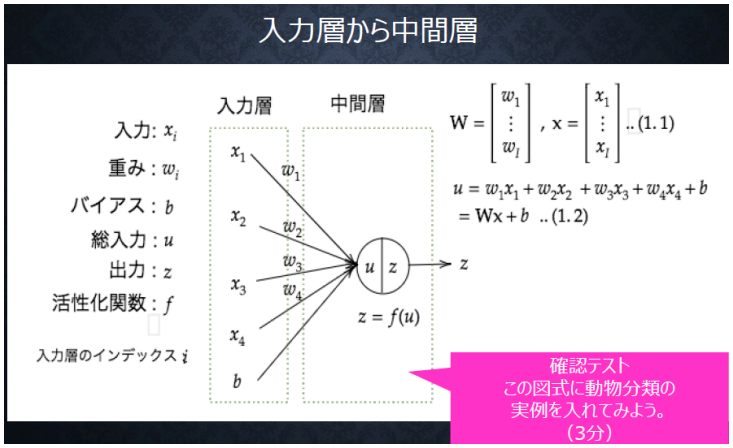
Q:ディープラーニングは結局何をやろうとしているのか。どの値の最適化が最終目的か。

入力データから求める出力データに変換できるようにモデルのパラメータ（重みとバイアス）を自動で調整（=学習）すること。

確認テスト 2



確認テスト 3



入力データ=動物を撮影した写真のピクセルデータ

確認テスト 4

```
import numpy as np

w1, w2, w3, w4 = 0.1, 0.2, 0.3, 0.4
x1, x2, x3, x4 = 1., 1., 1., 1.
b = 0.0
W = np.array([[w1], [w2], [w3], [w4]])
X = np.array([[x1], [x2], [x3], [x4]])
u = np.dot(W.T, X) + b
print(u)
```

```
[[1.]]
```

確認テスト 5

順伝播（3 層・複数ユニット）

```
# 1層の総出力
z1 = functions.relu(u1)

# 2層の総出力
z2 = functions.relu(u2)
```

実装演習

```
import numpy as np
from DNN_code_colab_day1.common import functions

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
# 順伝播（単層・単ユニット）

# 重み
W = np.random.randint(5, size=(2))
print_vec("重み", W)

# バイアス
b = np.random.random()
print_vec("バイアス", b)

# 入力値
x = np.array([2, 3])
print_vec("入力", x)

# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.relu(u)
print_vec("中間層出力", z)
```

```
# 結果
*** 重み ***
[3 4]
*** バイアス ***
0.6915546644179652
*** 入力 ***
[2 3]
*** 総入力 ***
18.691554664417964
*** 中間層出力 ***
18.691554664417964
```

考察

ニューラルネットワークの原型となったパーセプトロンの理論は 1950 年代に発表されたが、深層学習が本格的に実用化されたのは 2010 年代である。背景としてはこの間に計算リソースの向上といったコンピュータ性能が上がったことと学習データが大量に蓄積されていたからではないかと考えられる。

Section2:活性化関数

ノードへの入力信号の総和を出力信号に変換する関数のことを活性化関数と呼ぶ。活性化関数は入力信号の総和がどのように活性化（発火）するかということを決定する役割がある。重み付きの入力信号の総和を計算し、その和が活性化関数によって変換されるというのは数式で以下のように表せる。

$$a = \omega_1 x_1 + \omega_2 x_2 + b$$
$$y = h(a)$$

- ステップ関数
  入力が 0 を超えたら 1 を出力する。

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

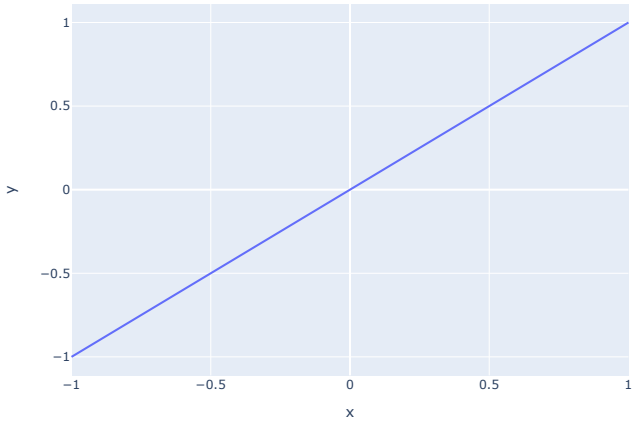
- シグモイド関数
  ニューラルネットワークでよく使われる関数。出力が 0 または 1 に近づくにつれ傾きが 0 になる。そのため 0 と 1 に偏ったデータ分布では誤差逆伝播法の勾配が小さくなる。これは勾配消失問題と呼ばれる。

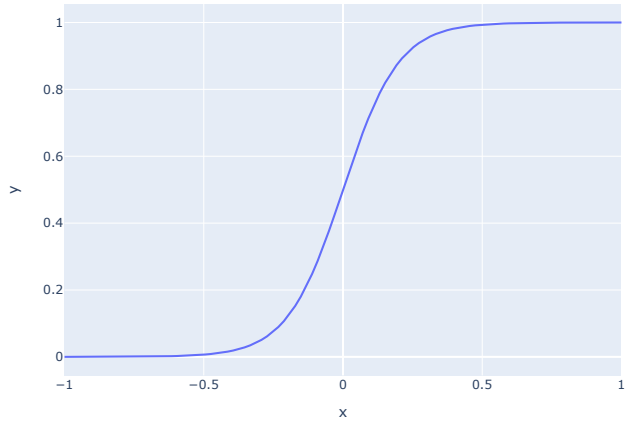
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

- ReLU（Rectified Linear Unit）
  入力が 0 を超えていたらその入力をそのまま返し、0 以下なら 0 を返す。スパース性がある。

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

確認テスト 1





関数fが線形である時には以下を満たす。

1. 加法性:任意のx, yに対して、$f(x+y) = f(x) + f(y)$
2. 斉次性:任意のx、スカラーλに対して、$f(\lambda x) = \lambda f(x)$

確認テスト 2

```
# 中間層出力
z = functions.sigmoid(u)
print_vec("中間層出力", z)
```

実装演習

```
import numpy as np
from DNN_code_colab_day1.common import functions

def print_vec(text, vec):
    print("*** " + text + " ***")
    print(vec)
# 順伝播（単層・複数ユニット）

# 重み
W = np.random.randint(5, size=(4, 3))
print_vec("重み", W)

# バイアス
b = np.array([0.1, 0.2, 0.3])
print_vec("バイアス", b)

# 入力値
x = np.array([1.0, 5.0, 2.0, -1.0])
print_vec("入力", x)

#  総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.sigmoid(u)
print_vec("中間層出力", z)
```

```
# 結果
*** 重み ***
[[3 3 4]
 [2 0 2]
 [2 0 4]
 [4 1 1]]
*** バイアス ***
[0.1 0.2 0.3]
*** 入力 ***
[ 1.  5.  2. -1.]
*** 総入力 ***
[13.1  2.2 21.3]
*** 中間層出力 ***
[0.99999795 0.90024951 1.        ]
```

考察

活性化関数に応じて適切な重みの初期値を選ぶ必要がある。

- sigmoid 関数、tanh 関数
  →「Xavier の初期値」
- ReLU 関数
  →「He の初期値」

Section3:出力層

ニューラルネットワークは分類問題と回帰問題の両方に用いることができる。出力層に関してはそのタスクに応じた活性化関数を選択する必要がある。

	回帰	二値分類	他クラス分類
活性化関数	恒等写像	シグモイド関数	ソフトマックス関数
誤差関数	二乗誤差	交差エントロピー	

- 恒等写像
  入力をそのまま出力する。
- ソフトマックス関数
  そのクラスに分類される確率を示す。

```
def softmax(a):
    c = np.max(a)  # オーバーフロー対策
    exp_a = np.exp(a - c)
    return exp_a / np.sum(exp_a)
```

  指数の計算をするとき、引数の値が大きいとオーバーフローする可能性がある。そのため softmax 関数に渡された引数で最大の値を差し引く処理を挟んでいる。c は分母分子でキャンセルアウトされるため出力に影響しない。
- 交差エントロピー誤差
  正解ラベルが 1 に対応する出力の自然対数を計算する。

```
def cross_entropy_error(y, t):
    d = 1e-7
    return -np.sum(t * np.log(y + d))
```

  t は正解ラベルで one-hot 表現である必要がある。d は$log(0)$の計算を回避するため。

確認テスト 1

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

符号差を考慮して 2 乗和をとる。1/2 は微分したときの係数をキャンセルアウトするため。

確認テスト 2

$$f(i, u) = \frac{e^{u_i}}{\sum_{k=1}^{K} e^{u_k}}$$

```
def softmax(x):
    if x.ndim == 2:                                 #入力xの次元が2なら
        x = x.T                                     #xを転置させ
        x = x - np.max(x, axis=1)                   #配列xの最大値で引く
        y = np.exp(x) / np.sum(np.exp(x), axis=0)   #②/③の計算
        return y.T                                  #再び転置させ返す
    x = x - np.max(x)  # オーバーフロー対策          #配列xの最大値で引く
    return np.exp(x) / np.sum(np.exp(x))            #②/③の計算を返す
```

確認テスト 3

$$E_n(w) = -\sum_{i=1}^{I} d_i \log y_i$$

```
def cross_entropy_error(d, y):
    if y.ndim == 1:                     #入力yの次元が1なら
        d = d.reshape(1, d.size)        #入力dを2次元に変換
        y = y.reshape(1, y.size)        #入力yを2次元に変換
    # 教師データがone-hot-vectorの場合、正解ラベルのインデックスに変換
    if d.size == y.size:                #dとyが同じ大きさの配列なら
        d = d.argmax(axis=1)            #dの行方向で一番大きい値のインデックスを代入

    batch_size = y.shape[0]             #yの行数を代入
    return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size
                                        #正解ラベルに対応するyの値を取得し②の計算をする
                                        #最後にバッチサイズで平均する
```

実装演習

```
# 多クラス分類


def init_network(input_size: int, hidden_size: int, output_size: int) -> dict:
    print("##### ネットワークの初期化 #####")

    network = {}
    network['W1'] = np.random.rand(input_size, hidden_size)
    network['W2'] = np.random.rand(hidden_size, output_size)
    network['b1'] = np.random.rand(hidden_size)
    network['b2'] = np.random.rand(output_size)

    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



def forward(network, x):

    print("##### 順伝播開始 #####")
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 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

    # 出力値
    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)))

    return y, z1

# 入力値
```

```
x = np.array([1., 2., 3.])
# 目標出力
d = np.array([0, 0, 0, 1, 0, 0])
# ネットワークの初期化
input_layer_size = 3
hidden_layer_size = 5
output_layer_size = 6
network = init_network(input_layer_size, hidden_layer_size, output_layer_size)

# 出力
y, z1 = forward(network, x)
# 誤差
loss = functions.cross_entropy_error(d, y)

# 表示
print("\n##### 結果表示 #####")
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("交差エントロピー誤差", loss)
```

```
#結果
##### ネットワークの初期化 #####
*** 重み1 ***
[[0.65185593 0.61783815 0.65618437 0.17001171 0.91929471]
 [0.9645046  0.24218416 0.43222158 0.73825968 0.1115167 ]
 [0.67421734 0.94860998 0.46684688 0.34572872 0.29138966]]
*** 重み2 ***
[[0.34092631 0.83664867 0.38053219 0.78345288 0.3710336  0.48325922]
 [0.77639755 0.10304212 0.98995038 0.73330654 0.97831841 0.37334055]
 [0.05110077 0.05552997 0.34178351 0.63990519 0.45536779 0.72549272]
 [0.78322786 0.01154058 0.36928999 0.48003021 0.14457613 0.78056733]
 [0.42227951 0.50427782 0.40788063 0.07742269 0.88411554 0.63592993]]
*** バイアス1 ***
[0.58396708 0.5215466  0.63804672 0.17060858 0.16649218]
*** バイアス2 ***
[0.22051696 0.18513746 0.98114737 0.58562907 0.69020381 0.48970382]
##### 順伝播開始 #####
*** 総入力1 ***
[5.18748422 4.46958302 3.55921486 2.85432582 2.18298927]
*** 中間層出力1 ***
[5.18748422 4.46958302 3.55921486 2.85432582 2.18298927]
*** 総入力2 ***
[ 8.79853786  6.31721126 10.54077146 11.74408854 10.95104403 10.86368614]
*** 出力1 ***
[0.02363635 0.00197676 0.13496552 0.44959018 0.20342389 0.1864073 ]
出力合計: 0.9999999999999998

##### 結果表示 #####
*** 出力 ***
[0.02363635 0.00197676 0.13496552 0.44959018 0.20342389 0.1864073 ]
*** 訓練データ ***
[0 0 0 1 0 0]
*** 交差エントロピー誤差 ***
0.7994186084673732
```

考察

　ニューラルネットワークが認識精度ではなく損失関数を設定する理由は認識精度だと不連続値で微分が 0 になる場合があるからである。例えば 100 枚の画像から 30 枚を正確に判別できたとすると認識精度は 30%となるが、重みを変化させても不連続値のため 30%のままで変化が現れないかもしれない。損失関数は連続な値をとるので微小な重みの変化も反映される。

Section4:勾配降下法

　ニューラルネットワークは学習時に最適なパラメータ（重みとバイアス）を探索する。最適なパラメータとは損失関数が最小値を取るときのパラメータの値である。損失関数の勾配をうまく利用して

最小値を探索することを勾配降下法と呼ぶ。勾配法では現在の場所から勾配方向に行っての距離だけ進む。その移動先でも同様に勾配を求めまた移動することを繰り返して損失関数の値を減らしてゆく。

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

ここでηは学習率と呼ばれる更新量である。学習率は大きすぎても小さすぎても最適解にたどり着くことはできないため、最適な値を選択する必要がある。

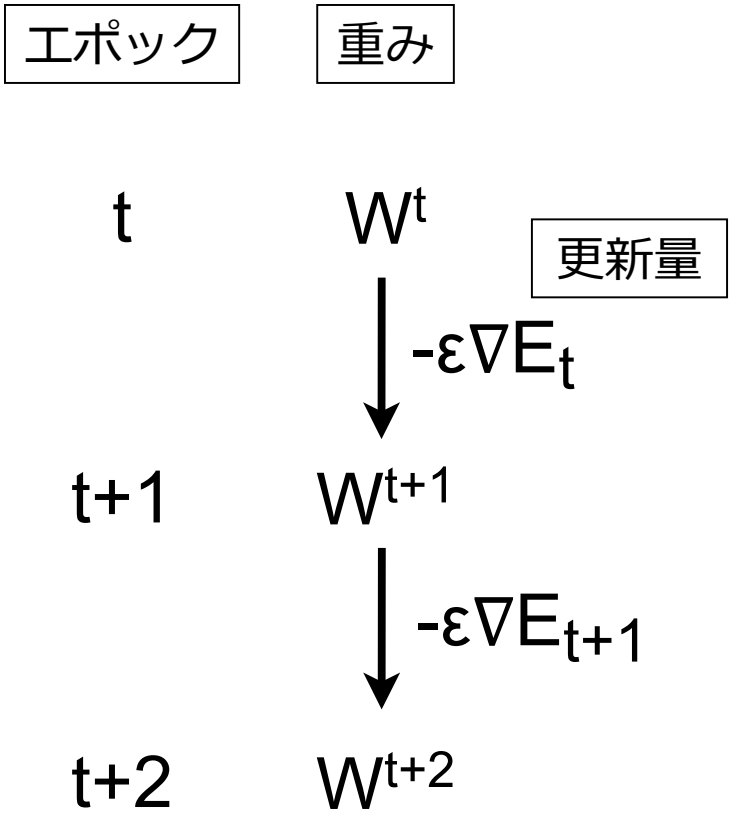
確認テスト 1

```
# 確率勾配降下法
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]
```

確認テスト 2

Q:オンライン学習とは何か
データをランダムに 1 レコードずつ学習させる方法。対してバッチ学習では一度にすべてのデータを使って学習を行う。オンライン学習はバッチ学習に比べてメモリ消費が少なくて済むが計算速度は劣る。また、データをある程度の塊に分割して学習を行う方法をミニバッチ学習と言う。

確認テスト 3


エポック
重み
t
W^t
更新量
$-\varepsilon\nabla E_t$
t+1
W^{t+1}
$-\varepsilon\nabla E_{t+1}$
t+2
W^{t+2}

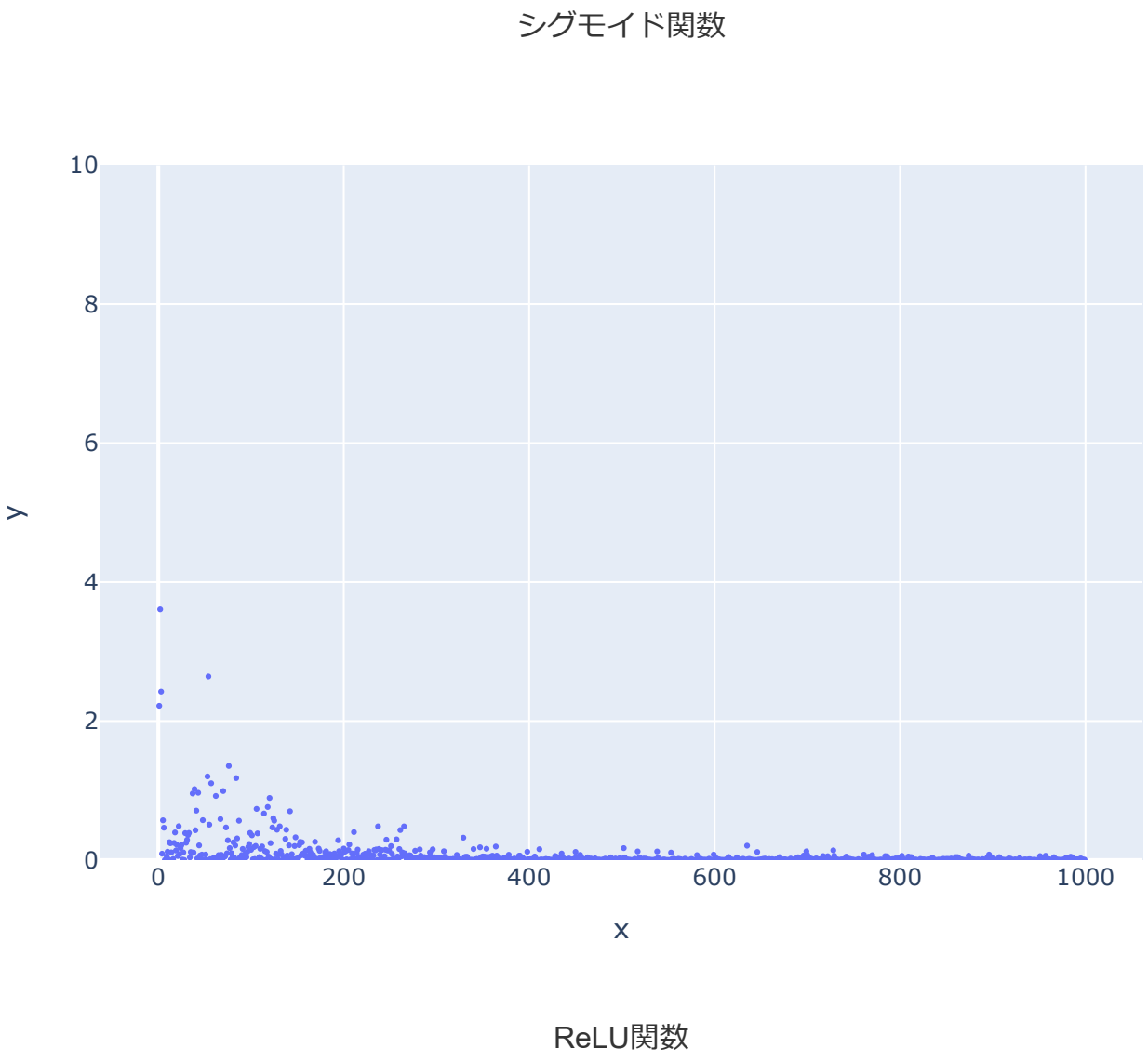
実装演習

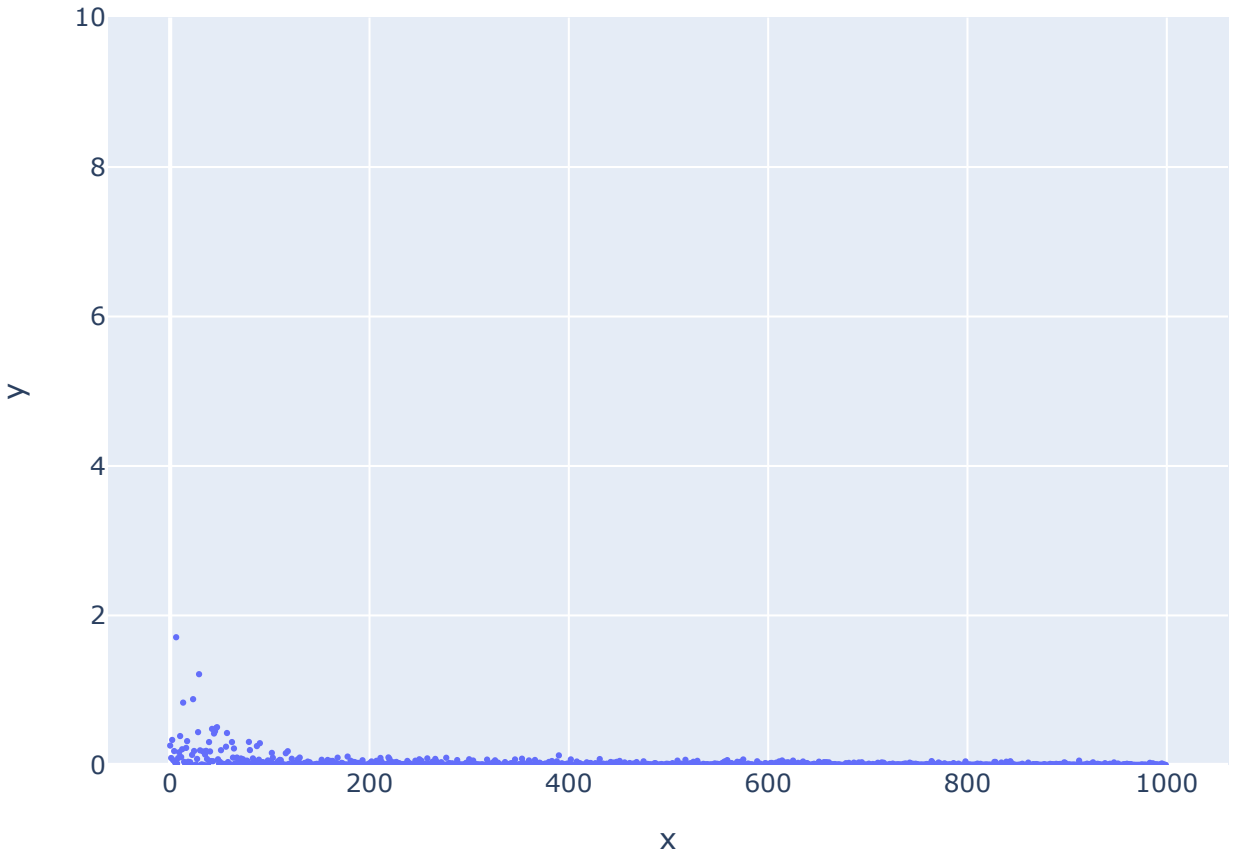
```
# yの値を予想するAI


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


def init_network():
    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()

    return network


def forward(network, x):

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    u1 = np.dot(x, W1) + b1
    # z1 = functions.sigmoid(u1)
    z1 = functions.relu(u1)
    u2 = np.dot(z1, W2) + b2
    y = u2

    return z1, y


def backward(x, d, z1, y):

    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)

    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 = []
learning_rate = 0.07
epoch = 1000
network = init_network()
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)
fig = px.scatter(x=lists, y=losses)
fig.update_traces(marker={"size": 3})
```

```
fig.show()
fig.write_image("grad_result.svg")
```

結果



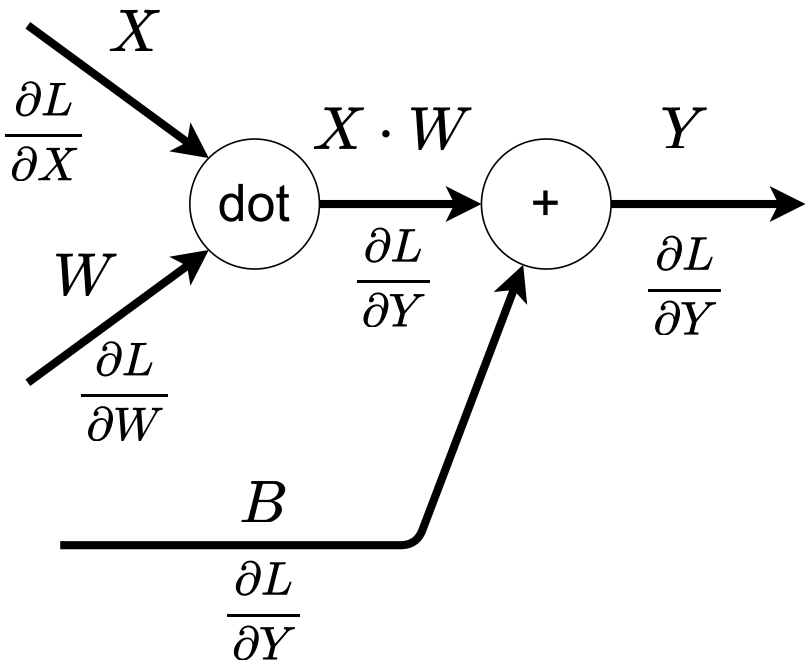


考察

活性化関数をシグモイド関数と ReLU 関数で比較してみたが ReLU の方が少ないエポックで誤差が小さくなっているように見える。どの活性化関数を選択するかも学習に重要だと感じた。

Section5:誤差逆伝播法

　出力層から入力層に向かって誤差を逆伝播し、各レイヤの重みとバイアスに対する勾配を計算することでネットワーク内のすべてのパラメータに対する損失関数の勾配を効率的に求めることができる。各レイヤの勾配は連鎖率を活用して計算できる。



確認テスト 1

```
# b2の勾配
grad['b2'] = np.sum(delta2, axis=0)
# W2の勾配
grad['W2'] = np.dot(z1.T, delta2)
# b1の勾配
grad['b1'] = np.sum(delta1, axis=0)
# W1の勾配
grad['W1'] = np.dot(x.T, delta1)
```

確認テスト 2

$$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}$$

```
# 出力層でのデルタ
delta2 = functions.d_mean_squared_error(d, y)
```

$$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial \omega_{ji}^{(2)}}$$

```
# W2の勾配
grad['W2'] = np.dot(z1.T, delta2)
```

考察

　ニューラルネットワークの核心とも言える誤差逆伝播法を学ぶことができた。最近は俄かに金子勇氏の ED 法が注目を集めている。こちらの理解も進めたい。

金子勇さんの ED 法の解説と弱点、行列積を使用した効率的な実装