

ラビット・チャレンジ

<実装演習レポート 深層学習前編(day1, day2)>

メールアドレス : mtop.jp@gmail.com

受講者名 : 山崎 英和

受講種別 : ラビット・チャレンジ

■ 深層学習前編(day1)

<Section1: 入力層～中間層>

◆ 要点まとめ

入力層：n 個の値 x_n を入力する層である（図 1 の①の部分）

中間層：入力層から出力した値 x_n それぞれに対して重み $w^{(l)}$ を乗算し、バイアスを $b^{(l)}$ を加算した値を入力とする l 個のノードから成る層である（図 1 の②の部分）

一つの間層ノードの入力 u_i は、 $u_i = Wx + b$ で表せる。ここで $W = \{w_1, w_2, \dots, w_n\}$ 、 $x = \{x_1, x_2, \dots, x_n\}$ 。

図 1：ニューラルネットワークの全体像（入力層、中間層）

【事前に用意する情報】

入力： $\mathbf{x}_n = [x_{n1} \dots x_{ni}]$

訓練データ： $\mathbf{d}_n = [d_{n1} \dots d_{nk}]$

【多層ネットワークのパラメータ】

$$\mathbf{w}^{(l)} \begin{cases} \text{重み: } \mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \dots & w_{1l}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{j1}^{(l)} & \dots & w_{jl}^{(l)} \end{bmatrix} \\ \text{バイアス: } \mathbf{b}^{(l)} = [b_1^{(l)} \dots b_j^{(l)}] \end{cases}$$

活性化関数： $\mathbf{f}^{(l)}(\mathbf{u}^{(l)}) = [f^{(l)}(u_1^{(l)}) \dots f^{(l)}(u_j^{(l)})]$

中間層出力： $\mathbf{z}^{(l)} = [z_1^{(l)} \dots z_k^{(l)}] = \mathbf{f}^{(l)}(\mathbf{u}^{(l)})$

総入力： $\mathbf{u}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$

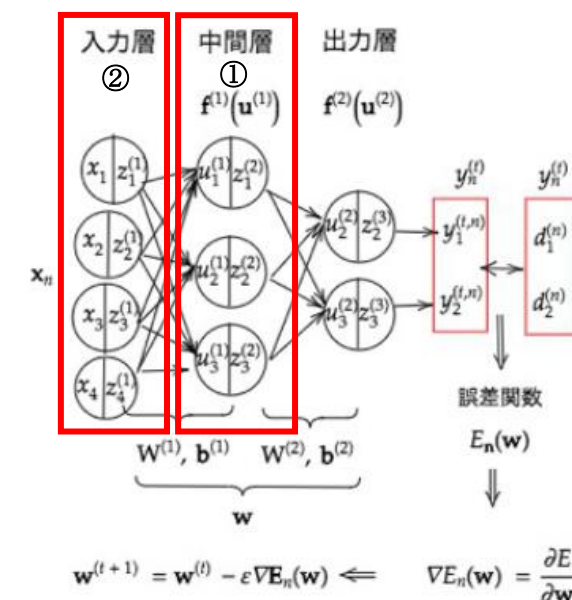
出力： $\mathbf{y}_n^{(l)} = [y_{n1}^{(l)} \dots y_{nk}^{(l)}] = \mathbf{z}^{(l)}$

誤差関数： $E_n(\mathbf{w})$

入力層ノードのインデックス： $i (= 1 \dots I)$

中間層ノードのインデックス： $j (= 1 \dots J)$

出力層ノードのインデックス： $k (= 1 \dots K)$



層のインデックス： $l (= 1 \dots L)$

訓練データのインデックス： $n (= 1 \dots N)$

試行回数のインデックス： $t (= 1 \dots T)$

◆実装演習結果キャプチャ

講義で説明された以下 2 つの実装演習結果を示す

<1_1_forward_propagation.ipynb>

1. 順伝播（単層・単ユニット）

▼ 順伝播（単層・単ユニット）

```
# 順伝播（単層・単ユニット）

# 重み
W = np.array([[0.1], [0.2]])

## 試してみよう...配列の初期化
#W = np.zeros(2)
#W = np.ones(2)
#W = np.random.rand(2)
#W = np.random.randint(5, size=(2))

print_veo("重み", W)

# バイアス
b = 0.5

## 試してみよう...数値の初期化
#b = np.random.rand() # 0~1のランダム数値
#b = np.random.rand() * 10 -5 # -5~5のランダム数値

print_veo("バイアス", b)

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

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

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

```
*** 重み ***
[[0.1]
 [0.2]]

*** バイアス ***
0.5

*** 入力 ***
[2 3]

*** 総入力 ***
[1.3]

*** 中間層出力 ***
[1.3]
```

2. 順伝播 (3層・複数ユニット)

▼ 順伝播 (3層・複数ユニット)

```
✓ 0 秒 # 順伝播 (3層・複数ユニット)

# 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 = {}

    # 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)
```

```

*** 入力 ***
[1. 2.]

##### ネットワークの初期化 #####
*** 重み1 ***
[[0.1 0.3 0.5]
 [0.2 0.4 0.6]]

*** 重み2 ***
[[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]

*** 重み3 ***
[[0.1 0.3]
 [0.2 0.4]]

*** バイアス1 ***
[0.1 0.2 0.3]

*** バイアス2 ***
[0.1 0.2]

*** バイアス3 ***
[1 2]

##### 順伝播開始 #####
*** 総入力1 ***
[0.6 1.3 2. ]

*** 中間層出力1 ***
[0.6 1.3 2. ]

*** 総入力2 ***
[1.02 2.29]

*** 出力1 ***
[0.6 1.3 2. ]

出力合計: 3.9

```

◆確認テストなどの考察結果

1-1. 順伝播（単層・単ユニット）に対し、重みとバイアスの初期値を変えた場合の総入力値を示す。

1-1-a. 重み初期値と総入力値の関係

以下の5種類の重み初期値と総入力値の関係を表 1-1-a に示す。

重みが大きいと総入力値が大きくなることがわかる。

- ① `np.array([[0.1],[0.2]])`
- ② `np.zeros(2)`
- ③ `np.ones(2)`
- ④ `np.random.rand(2)`
- ⑤ `np.random.randint(5, size=(2))`

表 1-1-a：重み初期値と総入力値の関係

	重み初期値①	重み初期値②	重み初期値③	重み初期値④	重み初期値⑤
重み	[[0.1] [0.2]]	[0. 0.]	[1. 1.]	[0.25306583 0.54214662]	[3 4]
バイアス	0.5	0.5	0.5	0.5	0.5
入力	[2 3]	[2 3]	[2 3]	[2 3]	[2 3]

総入力	[1.3]	0.5	5.5	2.6325715240 074987	18.5
-----	-------	-----	-----	------------------------	------

1-1-b. バイアス初期値と総入力値の関係

以下の 5 種類のバイアス初期値と総入力値の関係を表 1-1-b に示す。

重みのスケールに対してバイアスが大きいと、総入力結果がバイアスに依存するので注意が必要である。

- ① 0.5
- ② np.random.rand()
- ③ np.random.rand() * 10⁻⁵

表 1-1-b : バイアス初期値と総入力値の関係

	バイアス初期値①	バイアス初期値②	バイアス初期値③
重み	[[0.1] [0.2]]	[[0.1] [0.2]]	[[0.1] [0.2]]
バイアス	0.5	0.020577716555883 363	-4.045811801834844
入力	[2 3]	[2 3]	[2 3]
総入力	[1.3]	[0.82057772]	[-3.2458118]

1-2. 順伝播（3 層・複数ユニット）に対し、重みとバイアスの配列の形を示す。また、ランダム値を生成する場合のコードを以下に示す。

1-2-a. 重みとバイアスの初期値と総入力値の関係

以下の 3 種類の重みとバイアスの初期値と総入力値の関係を表 1-2-a に示す。

- ① default
- ② 重み初期値ランダム
 - 重み 1: np.random.rand(2,3)
 - 重み 2: np.random.rand(3,2)
 - 重み 3: np.random.rand(2,2)
- ③ バイアス初期値ランダム
 - バイアス 1: np.random.rand(3)
 - バイアス 2: np.random.rand(2)
 - バイアス 3: np.random.rand(2)*10⁻⁵

表 1-2-a : 重みとバイアスの初期値と総入力値の関係

	① Default	②重み初期値ランダム	③バイアス初期値ランダム
重み 1	[[0.1 0.3 0.5] [0.2 0.4 0.6]] shape: (2, 3)	[[0.50915371 0.72437543 0.58298486] [0.9136869 0.13617715 0.40058123]] shape: (2, 3)	[[0.1 0.3 0.5] [0.2 0.4 0.6]] shape: (2, 3)
重み 2	[[0.1 0.4] [0.2 0.5] [0.3 0.6]] shape: (3, 2)	[[0.5063469 0.62554928] [0.83408848 0.47562177] [0.26178949 0.85407209]] shape: (3, 2)	[[0.1 0.4] [0.2 0.5] [0.3 0.6]] shape: (3, 2)
重み 3	[[0.1 0.3] [0.2 0.4]] shape: (2, 2)	[[0.39863488 0.30860902] [0.35459681 0.35626496]] shape: (2, 2)	[[0.1 0.3] [0.2 0.4]] shape: (2, 2)
バイアス 1	[0.1 0.2 0.3] shape: (3,)	[0.1 0.2 0.3] shape: (3,)	[0.42111147 0.22173864 0.32262222] shape: (3,)
バイアス 2	[0.1 0.2] shape: (2,)	[0.1 0.2] shape: (2,)	[0.32782016 0.43990364] shape: (2,)
バイアス 3	[1 2] shape: (2,)	[1 2] shape: (2,)	[4.96346001 2.0662999] shape: (2,)
入力	[1. 2.]	[1. 2.]	[1. 2.]
総入力 1	[0.6 1.3 2.]	[2.4365275 1.19672973 1.68414732]	[0.92111147 1.32173864 2.02262222]
総入力 2	[1.02 2.29]	[2.77279867 3.73174196]	[1.2910657 2.68279088]

<Section2: 活性化関数>

◆要点まとめ

図 2 の中間層用の代表的な活性化関数として、ステップ関数、シグモイド関数、ReLU 関数がある。
これらの活性化関数の特徴を表 2-1 にまとめる。

図 2：ニューラルネットワークの全体像（活性化関数）

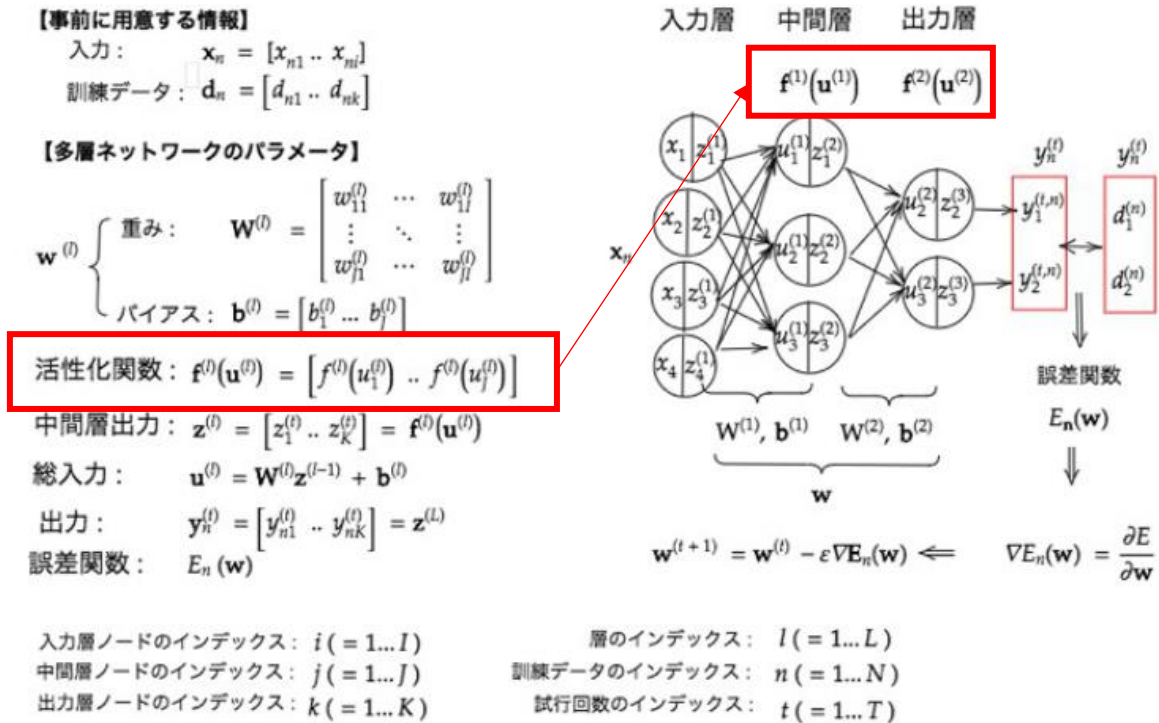
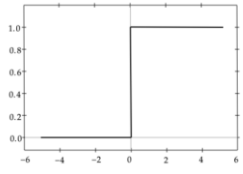
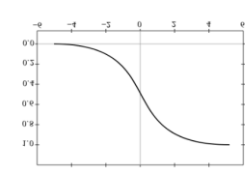
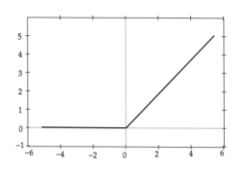


表 2-1: 中間層用の活性化関数の比較

	ステップ関数	シグモイド関数	ReLU 関数
概要	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。	0 ~ 1 の間に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。	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	$f(x) = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$	$f(u) = \frac{1}{1 + e^{-u}}$	$f(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leq 0) \end{cases}$

分布			
課題	0 -1 間の間を表現できず、線形分離可能なものしか学習できない	大きな値では出力の変化が微小なため、勾配消失問題を引き起こすことがある	-
サンプルコード ※ functions.py より	<pre>def step_function(x): if x > 0: return 1 else: return 0</pre>	<pre>def sigmoid(x): return 1/(1 + np.exp(-x))</pre>	<pre>def relu(x): return np.maximum(0, x)</pre>

◆実装演習結果キャプチャ

<1_1_forward_propagation.ipynb>

順伝播（単層・単ユニット）

活性化関数として以下の ReLU 関数を使用した時の実行結果を示す。

ReLU 関数は、入力値 x に対して、0 より小さいなら 0 を、0 以上なら x の値を返す関数である。

<functions.py>

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

実行結果：

```
##### 順伝播開始 #####
*** 入力 ***
[1. 2.]

*** 総入力1 ***
[1.2434647  1.81993732  1.72202739]

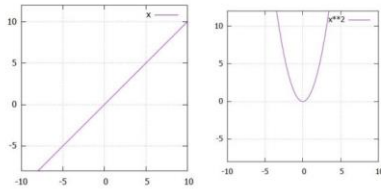
*** 中間層出力1 ***
[1.2434647  1.81993732  1.72202739]

*** 総入力2 ***
[1.44647849  3.03911354]

*** 出力1 ***
[1.2434647  1.81993732  1.72202739]

出力合計: 4.785429404673163
```

◆確認テストなどの考察結果

確認 テスト	問題	解答	考察 (コメント)
①	線形と非線形の違いを図に書いて簡易に説明せよ	<p>線形の図 非線形の図</p>  <p>より正確には、関数 f が線形 (linear) であるとは、</p> <ul style="list-style-type: none"> • 加法性 (additivity) 任意の x, y に対して、$f(x+y) = f(x) + f(y)$ • 斉次性 (homogeneity) 任意の x、任意のスカラー k に対して、$f(kx) = kf(x)$ <p>の両方が成立することです。</p>	非線形な関数は加法性や斉次性を満たさない
②	全結合 NN・単層・複数ノード配布されたソースコードより活性化関数が該当する箇所を抜き出せ	<p>1_1_forward_propagation.ipynb 順伝播 (単層・単ユニット) より</p> <pre># 1層の総出力 z1 = functions.relu(u1) # 2層の総出力 z2 = functions.relu(u2)</pre>	入力値 x に対して、0 より小さい場合は 0 を、0 以上なら x の値を返す関数

<Section3: 出力層>

◆要点まとめ

図 3：ニューラルネットワークの全体像（出力層）

【事前に用意する情報】

入力: $\mathbf{x}_n = [x_{n1} \dots x_{ni}]$

訓練データ: $\mathbf{d}_n = [d_{n1} \dots d_{nk}]$

【多層ネットワークのパラメータ】

$$\mathbf{w}^{(l)} \begin{cases} \text{重み: } \mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \dots & w_{1l}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{j1}^{(l)} & \dots & w_{jl}^{(l)} \end{bmatrix} \\ \text{バイアス: } \mathbf{b}^{(l)} = [b_1^{(l)} \dots b_j^{(l)}] \end{cases}$$

活性化関数: $\mathbf{f}^{(l)}(\mathbf{u}^{(l)}) = [f^{(l)}(u_1^{(l)}) \dots f^{(l)}(u_j^{(l)})]$

中間層出力: $\mathbf{z}^{(l)} = [z_1^{(l)} \dots z_k^{(l)}] = \mathbf{f}^{(l)}(\mathbf{u}^{(l)})$

総入力: $\mathbf{u}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$

出力: $\mathbf{y}_n^{(l)} = [y_{n1}^{(l)} \dots y_{nk}^{(l)}] = \mathbf{z}^{(l)}$

誤差関数: $E_n(\mathbf{w})$

入力層ノードのインデックス: $i (= 1 \dots I)$

中間層ノードのインデックス: $j (= 1 \dots J)$

出力層ノードのインデックス: $k (= 1 \dots K)$

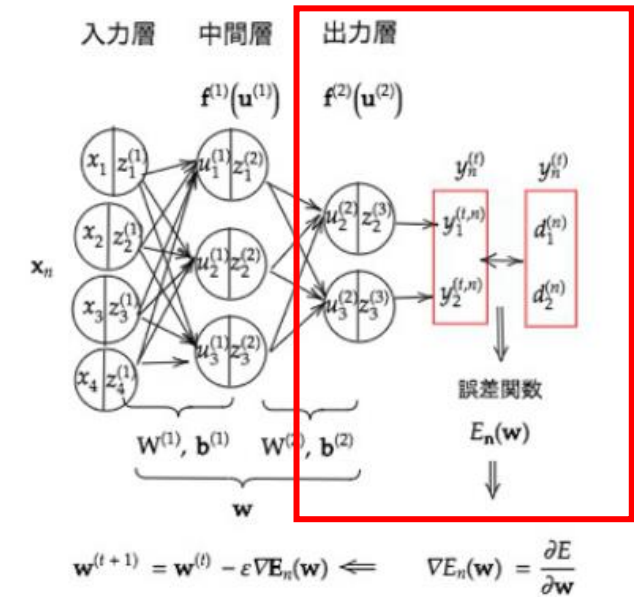


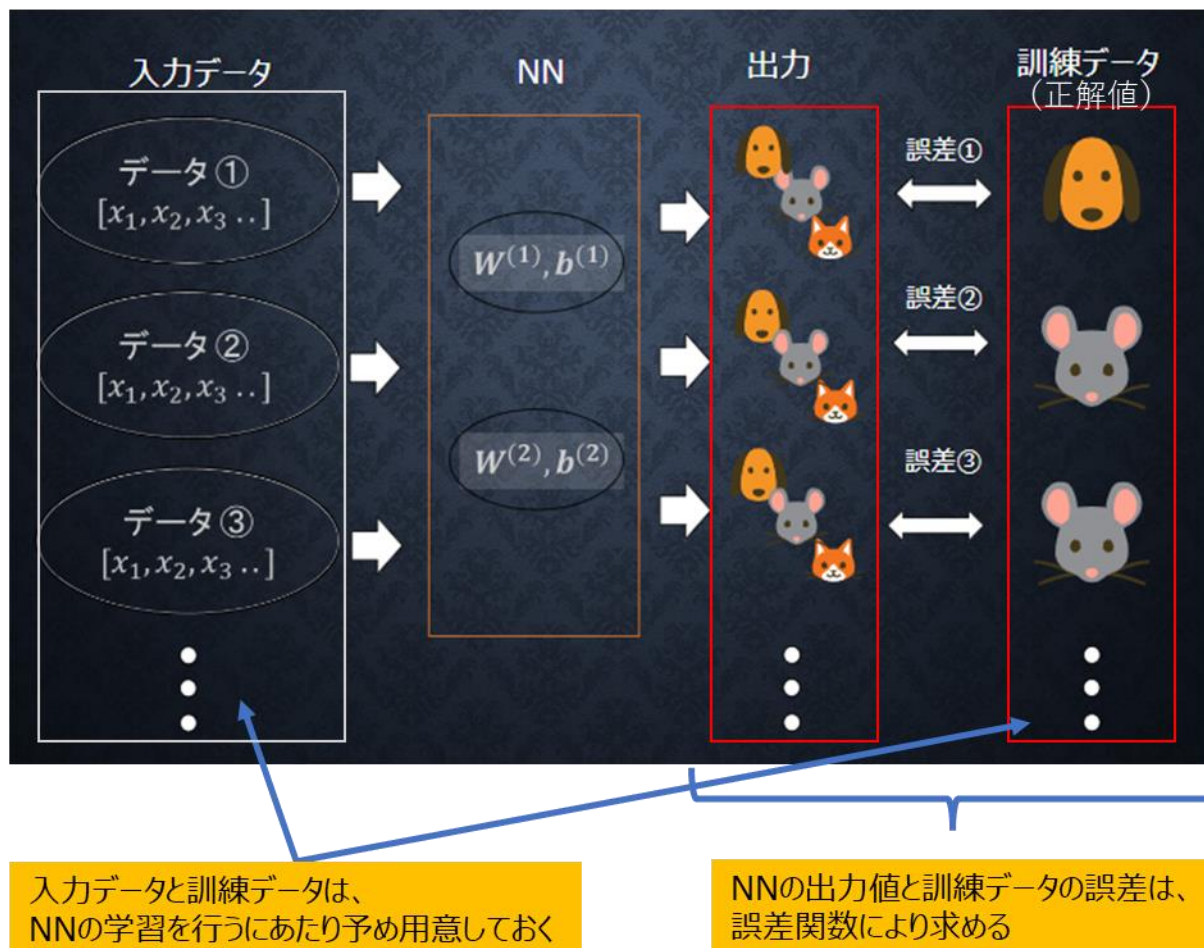
図 3 の出力層は、人間が使いやすい値を出力する層である。例えば分類問題ならば、分類クラスに属する確率を 0～1 の範囲で出力するような活性化関数を用いる。

出力層の出力結果は、訓練データ（正解値）と比較して誤差を得ることで予測精度の判定を行う。誤差を計算するために誤差関数を用いる。

この誤差を最小にするようにニューラルネットワークの学習を行うことで、精度の良い予測が得られるニューラルネットワークを作成することが出来る（図 3-2）。

図 3-2：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(NN)は、NNの出力値と訓練データの誤差を小さくするように学習することで、予測精度の高いNNを作成することが出来る



3-1. 誤差関数

代表的な誤差関数として、回帰問題では二乗誤差を、分類問題ではクロスエントロピー誤差を用いる。これらの誤差関数の特徴を表 3-1 にまとめる。

表 3-1: 誤差関数の比較

	二乗誤差 (L2 損失)	クロスエントロピー誤差
概要	<p>各データに対して「出力値と訓練データの差 (= 誤差)」の二乗値を計算し、その総和を 2 で割った値を出力する関数</p> <p>※二乗するのは出力値と訓練データの大小によるプラスとマイナスの誤差により打消しが発生し、誤差の大きさが正しく求まらないことを防止するため</p> <p>※2 で割るのは、誤差を最小化する値を求</p>	<p>出力値と訓練データの 2 つの確率分布の間に定義される尺度である。</p> <p>訓練データの確率分布が d であり、出力値の確率分布が y である場合に、とりうる複数の事象の中からひとつの事象を特定するために必要となるビット数の平均値を出力する関数</p> <p>確率分布 d と y が近似すると必要なビット数が小さくなり、d と y が近似しなくなると数値が大きくなることを利用</p>

	める誤差逆伝搬の計算で、微分により最小値を求めるときの計算を簡単にするため	する。
数式	$E_n(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^I (\mathbf{y}_n - \mathbf{d}_n)^2$	$E_n(\mathbf{w}) = - \sum_{i=1}^I d_i \log y_i$
サンプルコード	<pre># 平均二乗誤差 def mean_squared_error(d, y): return np.mean(np.square(d - y)) / 2</pre> <p>※functions.py より</p>	<pre># クロスエントロピー def cross_entropy_error(d, y): if y.ndim == 1: d = d.reshape(1, d.size) y = y.reshape(1, y.size) # 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 = d.argmax(axis=1) batch_size = y.shape[0] return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size</pre>

3-2. 活性化関数

図 3 の出力層用の代表的な活性化関数として、恒等写像、シグモイド関数（ロジスティック関数）、ソフトマックス関数がある。

これらの活性化関数の特徴を表 3-2 にまとめる。

表 3-2: 出力層用の活性化関数の比較

	恒等写像	シグモイド関数	ソフトマックス関数
概要	中間層で出力した値を変換なしでそのまま使用する。 回帰の出力層用に使います。	シグモイド関数は、 $(-\infty, \infty) \rightarrow (0, 1)$ となるよう変換する単調増加関数です。 シグモイド関数は、ニューラルネットワークにおいて入力値 u を確率値に相当する出力値 $f(u)$ に変換する活性化関数として使用されます。二値分類の出力層用に使用します。	ソフトマックス関数は、複数值からなる入力値ベクトル $\mathbf{u} = \{\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_i\}$ がどのような値を取ろうとも、出力値ベクトルの各要素を 0 から 1 の間に変換し、全要素の合計が 1 となるように正規化します。 ソフトマックス関数は、ニューラルネットワークにおいて入力値ベクトルの各要素を確率値に相当する出力値ベクトルに変換する活性化関数として使用されます。多クラス分類の出力層用に使用します。
数式	$f(u) = u$	$f(u) = \frac{1}{1 + e^{-u}}$	$f(\mathbf{i}, \mathbf{u}) = \frac{e^{u_i}}{\sum_{k=1}^K e^{u_k}}$
サンプルコード ※	<pre>def identity(x): return (x)</pre>	<pre>def sigmoid(x): return 1/(1+np.exp(-x))</pre>	<pre>def softmax(x): if x.ndim == 2: x = x.T</pre>

functions.py より			<pre> x = x - np.max(x, axis=0) y = np.exp(x) / np.sum(np.exp(x), axis=0) return y.T x = x - np.max(x) # オーバーフロー対策 return np.exp(x) / np.sum(np.exp(x)) </pre>
--------------------	--	--	---

◆実装演習結果キャプチャ

3-1. 誤差関数

<1_1_forward_propagation.ipynb>

回帰（2-3-2 ネットワーク）

回帰問題では、ニューラルネットワーク出力値と訓練データの誤差関数として平均二乗誤差を使用する。

```

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

*** 誤差 ***
0.9711249999999999

```

平均二乗誤差の実装を、表 3-1:誤差関数の比較のサンプルコードに示す。

<1_1_forward_propagation.ipynb>

2 値分類（2-3-1 ネットワーク）

分類問題では、ニューラルネットワーク出力値と訓練データの誤差関数としてクロスエントロピー誤差を使用する。

```

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

*** 誤差 ***
0.13427195993720972

```

クロスエントロピー誤差の実装を、表 3-1:誤差関数の比較のサンプルコードに示す。

3-2. 活性化関数

<functions.py>

シグモイド関数

二値分類問題では、ニューラルネットワーク出力層の活性化関数としてシグモイド関数を使用する。

シグモイド関数の実装を、表 3-2: 出力層用の活性化関数の比較のサンプルコードに示す。

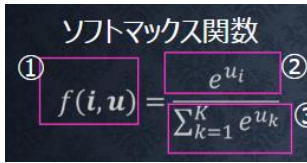
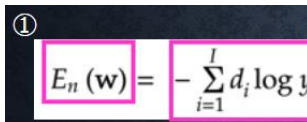
<functions.py>

ソフトマックス関数

三値以上の多値分類問題では、ニューラルネットワーク出力層の活性化関数としてソフトマックス関数を使用する。

ソフトマックス関数の実装を、表 3-2: 出力層用の活性化関数の比較のサンプルコードに示す。

◆確認テストなどの考察結果

確認 テ ス ト	問題	解答	考察 (コメント)
①	<p>・なぜ、引き算でなく二乗するか述べよ</p> <p>・下式の 1/2 はどういう意味を持つか述べよ</p> $E_n(\mathbf{w}) = \frac{1}{2} \sum_{j=1}^I (y_j - d_j)^2 = \frac{1}{2} \ \mathbf{y} - \mathbf{d}\ ^2$	<p>・二乗する理由 引き算を行うだけでは、各ラベルでの誤差で正負両方の値が発生し、全体の誤差を正しく表すのに都合が悪い。二乗してそれぞれのラベルでの誤差を正の値になるようにするため</p> <p>・1/2 する理由 実際にネットワークを学習するときに行う誤差逆伝搬の計算で、誤差関数の微分を用いるが、その際の計算式を簡単にするため。本質的な意味はない。</p>	<p>統計学で定義された数式を実装している。</p> <p>ニューラルネットワークの学習では何層ものノード（中間層）に対して、誤差関数により学習を行うため、一回の計算量を簡単にすることで全体の計算量を大幅に削減することが出来る</p>
②	<p>①～③の数式に該当するソースコードを示し、一行づつ処理の説明をせよ。</p> 	<pre>def softmax(x): if x.ndim == 2: x = x.T x = x - np.max(x, axis=0) y = np.exp(x) / np.sum(np.exp(x), axis=0) return y.T x = x - np.max(x) # オーバーフロー対策 return np.exp(x) / np.sum(np.exp(x))</pre> <p>・①～③の数式に該当する箇所は、上記の赤枠の箇所である</p> <p>・if 文はミニバッチを用いる場合の処理</p> <p>・オーバーフロー対策は安定して計算するため</p>	<p>ソフトマックス関数は、三値以上の多クラス分類時に使用する。</p> <p>各値を 0～1 の範囲に変換し、それらの総和は 1 となるように正規化することで、各クラスの発生確率を示す</p>
①	<p>①～②の数式に該当するソースコードを示し、一行づつ処理の説明をせよ。</p> 	<pre>def cross_entropy_error(d, y): if y.ndim == 1: d = d.reshape(1, d.size) y = y.reshape(1, y.size) # 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 = d.argmax(axis=-1) batch_size = y.shape[0] return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size</pre> <p>・①、②の数式に該当する箇所は、上記の赤枠の箇所である</p> <p>・コードの d が訓練データ、y が NN の出力</p> <p>・if 分はミニバッチを用いる場合の処理</p> <p>・+1e-7 は、y=0 の場合に$-\infty$の計算を避けるため、有限の値を与える工夫である</p>	<p>$d_i \log y_i$ は、訓練データ（教師データ）\mathbf{d} が、one-hot-vector（正解のラベルが 1 で、他は 0）であるため、numpy で実装すると、$\text{np.log}(y[\text{arange}(\text{batch_size}), d])$ となる。</p>

			これにより、正解ラベルに 対応する y の値に対数を とっている
--	--	--	--

<Section4: 勾配降下法>

◆要点まとめ

図 4：ニューラルネットワークの全体像（勾配降下法）

【事前に用意する情報】

入力: $\mathbf{x}_n = [x_{n1} \dots x_{ni}]$

訓練データ: $\mathbf{d}_n = [d_{n1} \dots d_{nk}]$

【多層ネットワークのパラメータ】

$$\mathbf{w}^{(l)} \begin{cases} \text{重み: } \mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \dots & w_{1j}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{i1}^{(l)} & \dots & w_{ij}^{(l)} \end{bmatrix} \\ \text{バイアス: } \mathbf{b}^{(l)} = [b_1^{(l)} \dots b_j^{(l)}] \end{cases}$$

活性化関数: $\mathbf{f}^{(l)}(\mathbf{u}^{(l)}) = [f^{(l)}(u_1^{(l)}) \dots f^{(l)}(u_j^{(l)})]$

中間層出力: $\mathbf{z}^{(l)} = [z_1^{(l)} \dots z_k^{(l)}] = \mathbf{f}^{(l)}(\mathbf{u}^{(l)})$

総入力: $\mathbf{u}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$

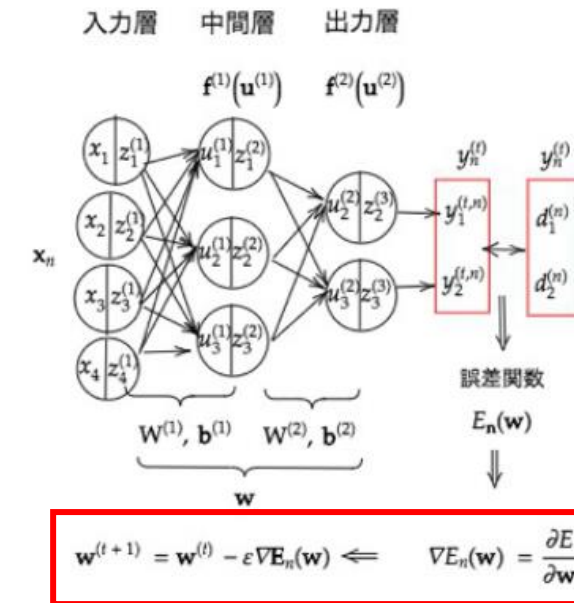
出力: $\mathbf{y}_n^{(l)} = [y_{n1}^{(l)} \dots y_{nk}^{(l)}] = \mathbf{z}^{(l)}$

誤差関数: $E_n(\mathbf{w})$

入力層ノードのインデックス: $i (= 1 \dots I)$

中間層ノードのインデックス: $j (= 1 \dots J)$

出力層ノードのインデックス: $k (= 1 \dots K)$



層のインデックス: $l (= 1 \dots L)$

訓練データのインデックス: $n (= 1 \dots N)$

試行回数のインデックス: $t (= 1 \dots T)$

図 4 に、ニューラルネットワークの学習で使用する勾配降下法を示す。

- ・ニューラルネットワークの学習とは、誤差と同意義の損失・コストを最小化する重みやバイアスのパラメータを決定することである。
- ・歴史的には、勾配降下法 → 確率勾配降下法 → バッチ勾配降下法の流れで、各種の学習効率を高める改善が行われているが、現在ではバッチ勾配降下法が使われるのが一般的である。
- ・アルゴリズムとしては、Adam が一般的に多く使われる。

これらの方式の特徴を表 4-1 にまとめる。

表 4-1: 勾配下降法の比較

	勾配降下法	確率的勾配降下法	バッチ勾配降下法
概要	全サンプルの平均誤差	ランダムに抽出したサンプルの誤差	ランダムに分割したデータの集合(ミニバッチ) D_t に属するサンプルの平均誤差
数式	$w^{(t+1)} = w^{(t)} + \epsilon \nabla E$ ※ ϵ : 学習率	$w^{(t+1)} = w^{(t)} + \epsilon \nabla E_n$	$w^{(t+1)} = w^{(t)} + \epsilon \nabla E_t$ $E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n$ $N_t = D_t $
メリット	—	・データが冗長な場合の計	・確率的勾配降下法のメ

		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	リットを損なわず、計算機の計算資源を有効活用できる →CPU によるスレッド並列化、GPU を利用した SIMD 化
デメリット	・学習率の値によって学習の効果が大きく異なる ・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 ・学習率が小さい場合、発散することはないが、小さすぎると収束するまでに時間がかかってしまう。また、局所極小解にはまりやすい。	大きな値では出力の変化が微小なため、勾配消失問題を引き起こす事がある	—
アルゴリズム	・Momentum ・AdaGrad ・Adadelta ・Adam ※オンライン学習：学習データが入ってくるたびに都度パラメータを更新し学習を進める方法 ※バッチ学習：一度にすべての学習データを使って、パラメータの更新を行う		

◆実装演習結果キャプチャ

<1_3_stochastic_gradient_descent.ipynb>

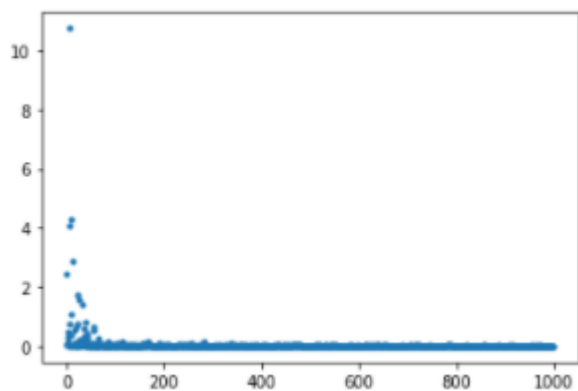
確率的勾配降下法

以下は、確率的勾配降下法におけるエポックごとの誤差（loss）の散布図である。

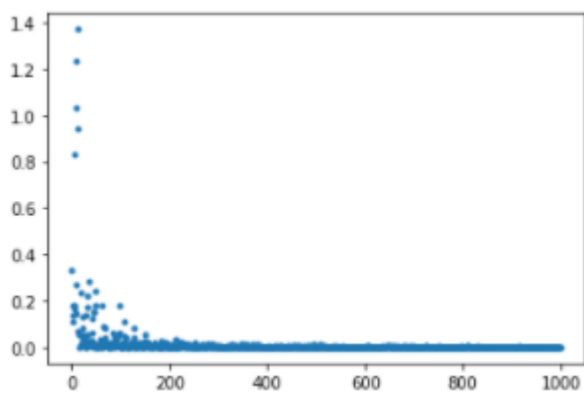
複数回実行してランダムに抽出したサンプルにより誤差を最小化できていることを確認した。

※y 軸：誤差、x 軸：エポック数

試行 1：



試行 2 :



◆確認テストなどの考察結果

確認テスト	問題	解答	考察 (コメント)
①	<p>該当するソースコードを探してみよう</p> <p>【勾配降下法】</p> $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$ $\nabla E = \frac{\partial E}{\partial \mathbf{w}} = \left[\frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_M} \right]$ <p>ϵ: 学習率</p>	<p><1_3_stochastic_gradient_descent.ipynb></p> <p>確率的勾配降下法</p> <p>赤枠の箇所が該当する。</p> <p>各重み $W1, W2$、バイアス $b1, b2$ ごとに、勾配降下法でパラメータを更新している。</p> <p><code>grad[key]</code>がそれぞれのパラメータに対する誤差∇Eであり、<code>learning_rate</code> が学習率ϵである。</p>	<p>確率的勾配降下法の実装例であるが、赤枠の箇所は勾配下降法と共通である</p>

		<pre> # 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) </pre>	
②	オンライン学習とは何か 2 行でまとめよ	学習データが入るたびに都度パラメータを更新し、学習を進めていく方法	一方、バッチ学習では、一度にすべての学習データを使って、パラメータの更新を行う
③	この数式の意味を図に書いて説明せよ $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \varepsilon \nabla E_t$	<p>エポック $t, t+1, t+2$ と学習を進めるごとに、各エポックでの誤差 $\nabla E(t)$ に学習率 ε を掛けたものでパラメータを更新する</p>	学習データをランダムに分割したミニバッチに分割し、そのミニバッチの予測結果と訓練データの誤差からパラメータを更新する

<Section5: 誤差逆伝播法>

◆要点まとめ

図5：ニューラルネットワークの全体像（誤差逆伝播法）

【事前に用意する情報】

入力: $\mathbf{x}_n = [x_{n1} \dots x_{ni}]$

訓練データ: $\mathbf{d}_n = [d_{n1} \dots d_{nk}]$

【多層ネットワークのパラメータ】

$$\mathbf{w}^{(l)} \begin{cases} \text{重み: } \mathbf{W}^{(l)} = \begin{bmatrix} w_{11}^{(l)} & \dots & w_{1J}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{I1}^{(l)} & \dots & w_{IJ}^{(l)} \end{bmatrix} \\ \text{バイアス: } \mathbf{b}^{(l)} = [b_1^{(l)} \dots b_J^{(l)}] \end{cases}$$

活性化関数: $\mathbf{f}^{(l)}(\mathbf{u}^{(l)}) = [f^{(l)}(u_1^{(l)}) \dots f^{(l)}(u_J^{(l)})]$

中間層出力: $\mathbf{z}^{(l)} = [z_1^{(l)} \dots z_J^{(l)}] = \mathbf{f}^{(l)}(\mathbf{u}^{(l)})$

総入力: $\mathbf{u}^{(l)} = \mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}$

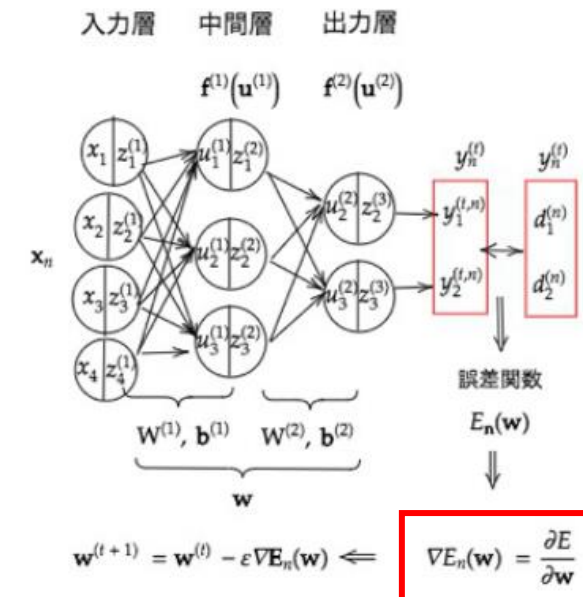
出力: $\mathbf{y}_n^{(l)} = [y_{n1}^{(l)} \dots y_{nK}^{(l)}] = \mathbf{z}^{(L)}$

誤差関数: $E_n(\mathbf{w})$

入力層ノードのインデックス: $i (= 1 \dots I)$

中間層ノードのインデックス: $j (= 1 \dots J)$

出力層ノードのインデックス: $k (= 1 \dots K)$



層のインデックス: $l (= 1 \dots L)$

訓練データのインデックス: $n (= 1 \dots N)$

試行回数のインデックス: $t (= 1 \dots T)$

図 5 に、ニューラルネットワークの学習で使用する誤差逆伝搬法の範囲を示す。

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

を計算する方法として、一般的に以下の数値微分で求めることが出来るが、各パラメータ w_M について、 $E(w_m + h)$ や $E(w_m - h)$ 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、計算負荷が大きい

$$\frac{\partial E}{\partial w_m} \approx \frac{E(w_m + h) - E(w_m - h)}{2h} \quad \text{※ } h \text{ は微小な値}$$

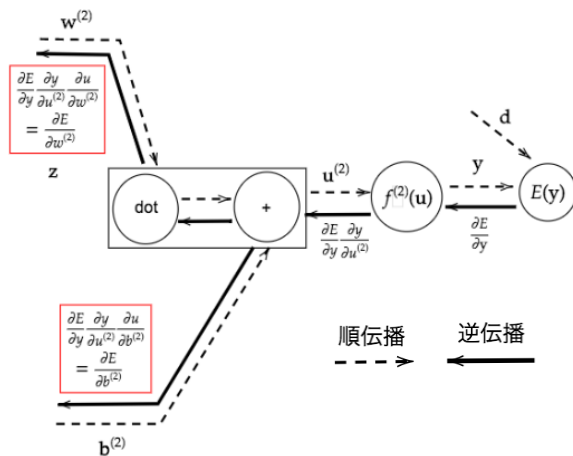
そこで、算出された誤差を、出力層側から順に微分し、前の層、前の層へと微分し、最小限の計算で各パラメータでの微分値を解析的に計算する手法である誤差逆伝播法で求める。

【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、前の層へと伝播させる。
- ・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である。
- ・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。

例えば、図 5-1 の例では、 $w^{(2)}$ に対する $\frac{\partial E}{\partial w^{(2)}}$ は、 $\frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial w^{(2)}}$ と出力結果から逆に微分した結果を掛け合わせることで求まる。

図 5-1. 誤差逆伝播法



◆実装演習結果キャプチャ

<1_3_stochastic_gradient_descent.ipynb>

確率的勾配降下法

各層の入力値を x 、中間層の出力を $z1$ 、出力層の出力を y 、期待値を d とした場合、赤枠の箇所が誤差逆伝播法適用箇所となる

```
# 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]
```

backward 関数内では、出力層と中間層で活性化関数が異なるため、それぞれの勾配を計算している

```

# 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)

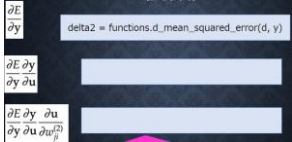
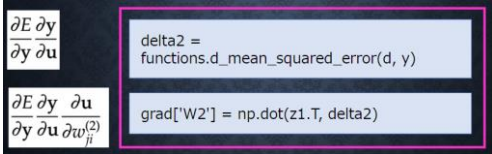
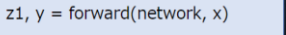
    # print_vec("偏微分_重み1", grad["W1"])
    # print_vec("偏微分_重み2", grad["W2"])
    # print_vec("偏微分_バイアス1", grad["b1"])
    # print_vec("偏微分_バイアス2", grad["b2"])

    return grad

```

◆確認テストなどの考察結果

確認テスト	問題	解答	考察 (コメント)
①	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。	<1_3_stochastic_gradient_descent.ipynb> 確率的勾配降下法 赤枠の箇所が該当する。	中間層と出力層で活性化関数が異なるため、それぞれの微分値を保持している

		<pre> # 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 </pre>	
②	<p>2つの空欄に該当するソースコードを探せ</p> 	<p><1_3_stochastic_gradient_descent.ipynb> 確率的勾配降下法 それぞれ以下となる</p>  <p>※ここで用いられるz1は以下のコードで生成される</p> 	<p>出力結果から逆に微分した結果を掛け合わせることで求まる</p>

■ 深層学習前編(day2)

<Section1: 勾配消失問題>

◆ 要点まとめ

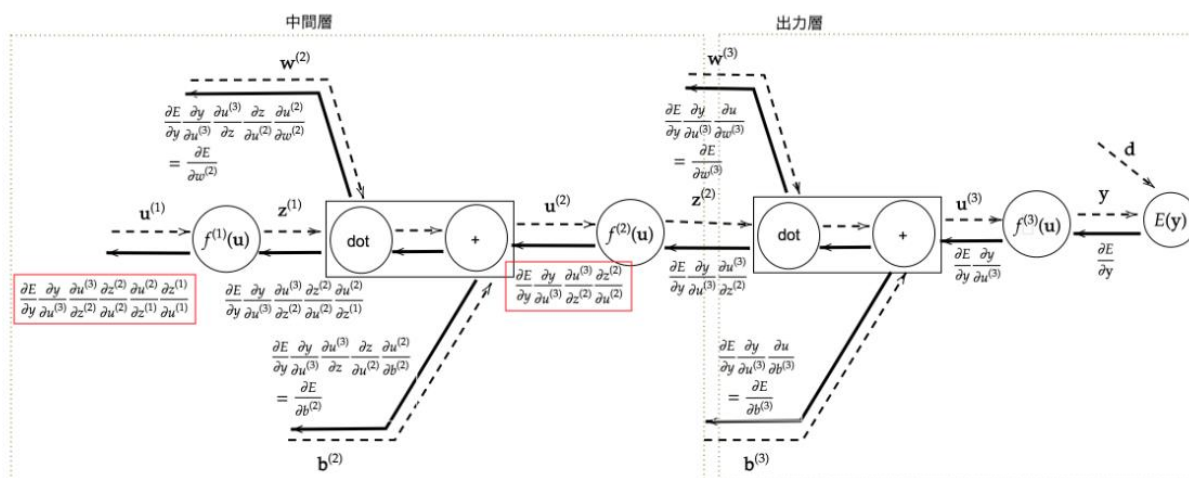
勾配消失問題とは

誤差逆伝播法が下位層に進んでいくに連れて、勾配がどんどん緩やかになっていく。そのため、勾配降下法による更新では下位層のパラメータはほとんど変わらず、エポック数を増やしても最適値が求まらない。

図 6 の例では、重みパラメータ $w^{(2)}$ に対する $\frac{\partial E}{\partial w^{(2)}}$ は、以下の複数の微分値の積で求まるため、各微分値が 1 より小さい値であると、中間層が増えるに従い 0 に近づき最適値を求めることが出来ない

$$\frac{\partial E}{\partial y} \frac{\partial y}{\partial u^{(3)}} \frac{\partial u^{(3)}}{\partial z} \frac{\partial z}{\partial u^{(2)}} \frac{\partial u^{(2)}}{\partial w^{(2)}} = \frac{\partial E}{\partial w^{(2)}}$$

図 6：誤差逆伝播法によるパラメータの学習



勾配消失問題の解決方法

- 活性化関数の選択
- 重みの初期値設定
- バッチ正規化

a) 活性化関数の選択

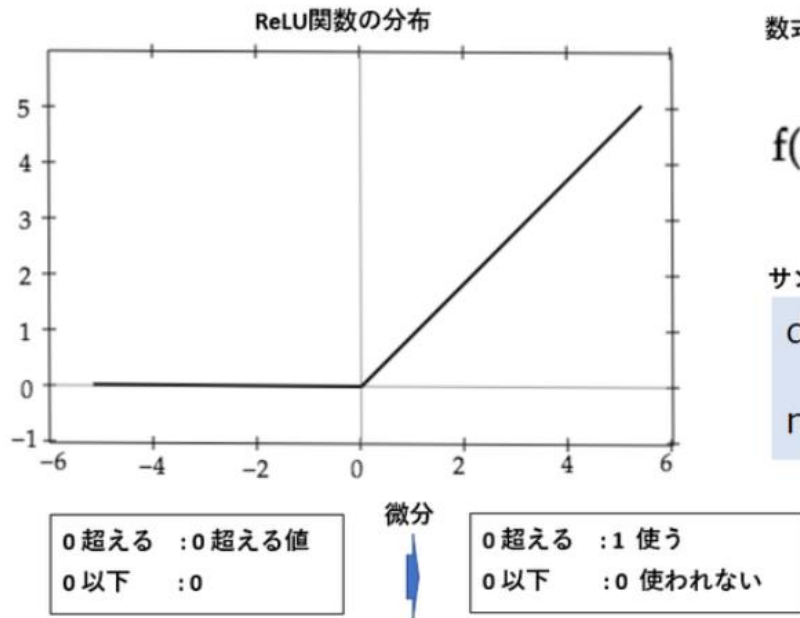
シグモイド関数から ReLU 関数へ

シグモイド関数は 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 状態しかないのに対し信号の強弱を伝えることが可能。±の値が大きくなると変化が微小になるため、勾配消失問題を引き起こすことがある。

一方、ReLU 関数は、入力値が 1 より大きい場合は微分値が 1 となり勾配消失問題を回避することが出来る。また、入力値 0 以下は常に微分値が 0 となるので、ニューロン群(中間層)の活性化をスパース (sparse : 疎) に出来る。

図 6-1. ReLU 関数と微分結果

ReLU関数



b) 重みの初期値設定

b-1) Xavier の初期値

重み w の初期値が 0 に近いと重みの更新量が小さくなり勾配消失が発生しやすい。そこで Xavier の初期値を使った乱数を利用する。

Xavier はシグモイド関数など S 字曲線的な関数へ適用する。

- ReLU 関数
- シグモイド (ロジスティック) 関数
- 双曲線正接関数

乱数発生確率分布には、平均 = 0、分散 = 1 の標準正規分布を使うと図 5-2 の 1 番目のグラフのように重みが 0 と 1 に偏り、その場合の微分値はほぼ 0 であるため勾配消失が発生する。そのため、重みの要素を前の層のノード数の平方根で除算した値とすることで、図 5-2 の 3 番目のグラフのような分布になるようにしている。

図 5-2 の 1 番目のグラフ : 標準正規分布で重みを初期化した場合は、各レイヤの出力は 0 と 1 に偏る。

シグモイド関数などは 0 と 1 の微分値はほとんど 0 であるため、勾配消失が発生する

図 5-2 の 2 番目のグラフ : 標準偏差を 0.01 にした正規分布で重みを初期化した場合は、出力値は

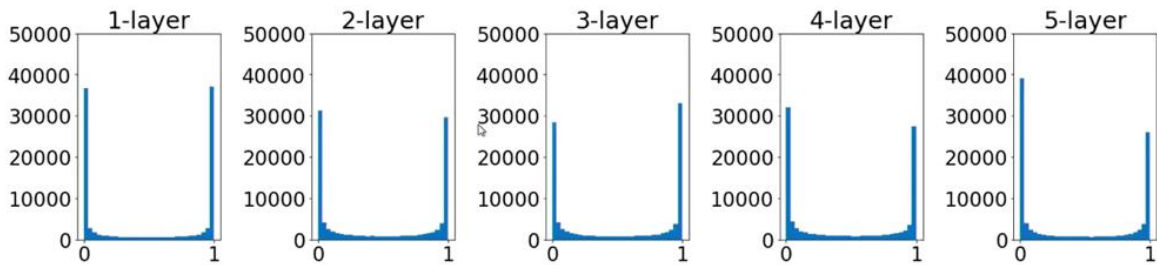
0.5 付近に偏るため、勾配消失は発生しづらくなる。一方で、ほとんどの値が 0.5 付近に集中するため、表現が大きく損なわれる

図 5-2 の 3 番目のグラフ : Xavier の初期値。各レイヤの出力値がある程度のばらつきを持ちつつ、0 や 1 に偏ることもないため、活性化関数の表現力を保ったまま、勾配消失への対策を取ることが出来る

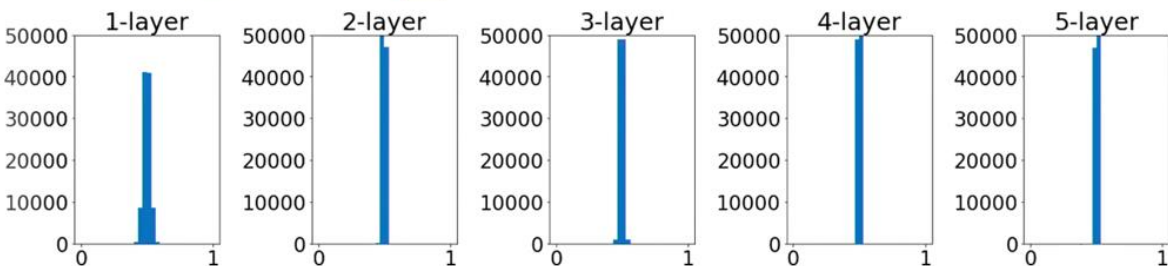
図 5-2. Xavier の初期値

重みの初期値設定-Xavier ※Sigmoid関数など、S字曲線なものへ適用

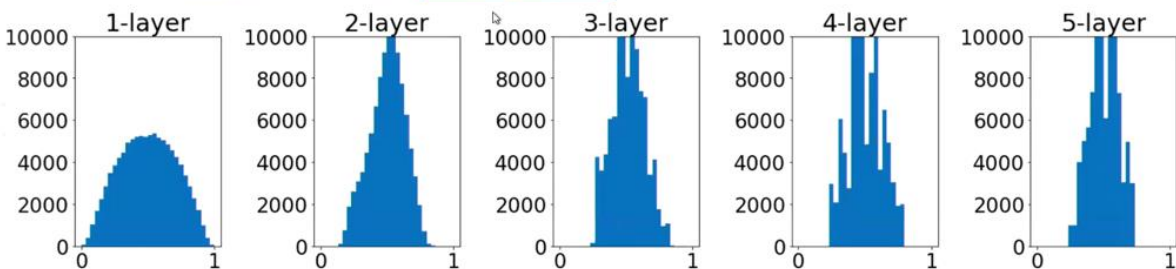
```
1 show_activation(sigmoid, lambda n: np.random.randn(n, n) * 1)
```



```
1 show_activation(sigmoid, lambda n: np.random.randn(n, n) * 0.01)
```



```
1 show_activation(sigmoid, lambda n: np.random.randn(n, n) * np.sqrt(1.0 / n), (0, 10000))
```



b-2) He の初期値

He は、ReLU 関数などの S 字曲線でない関数に適用する。重みの要素を前の層のノード数の平方根で除算した値に対し $\sqrt{2}$ で乗算した値で、図 5-3 の 3 番目のような分布になるようにしている。

図 5-3 の 1 番目のグラフ : 標準正規分布で重みを初期化した場合は、各レイヤの出力は 0 に偏り、表現力が全くなくなる

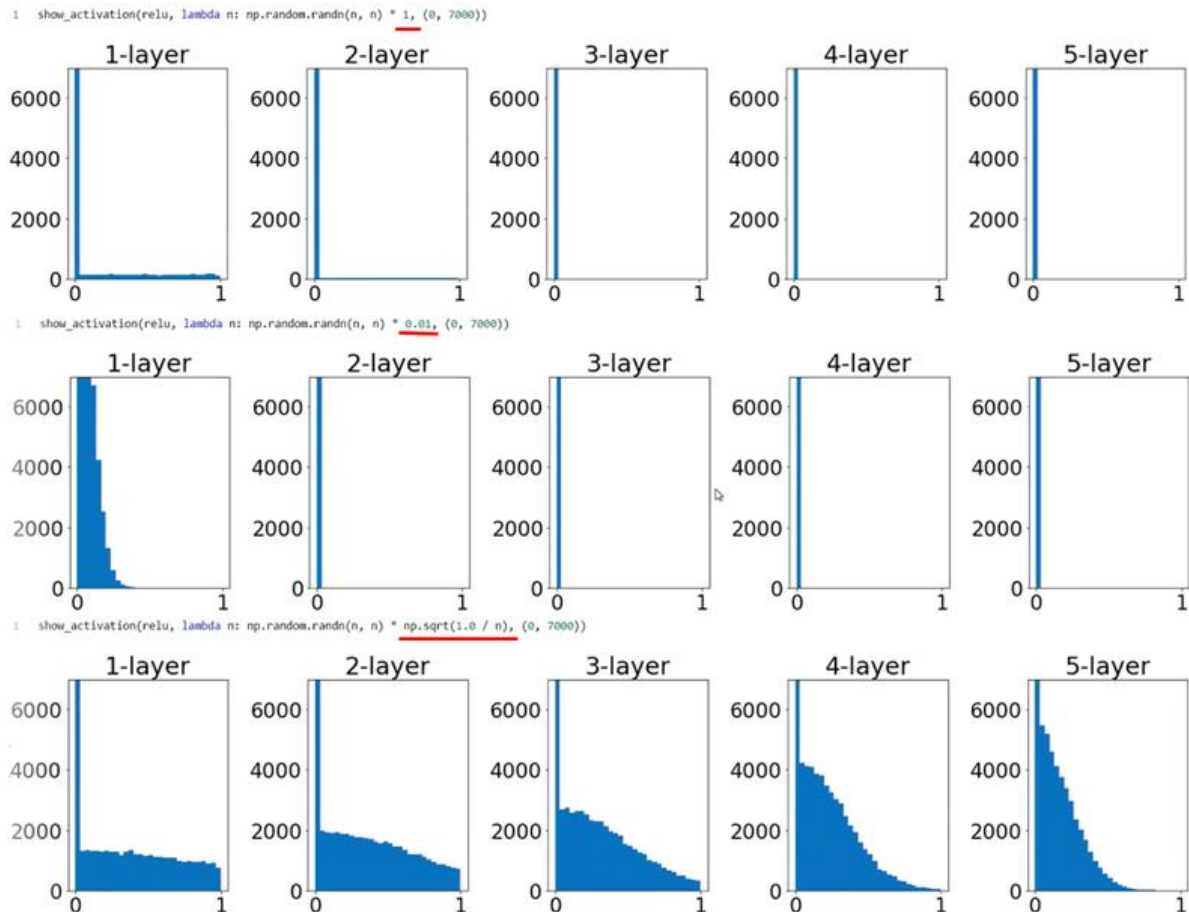
図 5-3 の 2 番目のグラフ : 標準偏差を 0.01 にした正規分布で重みを初期化した場合でも出力値は 0 に偏ったままである

図 5-3 の 3 番目のグラフ : He の初期値。各レイヤの出力値がある程度のばらつきを持ちつつ、0 に偏る

こともないため、活性化関数の表現力を保ったまま、勾配消失への対策を取ることが出来る

図 5-3. He の初期値

重みの初期値設定-He ※Relu関数など、s字曲線でないのに適用



c) バッチ正規化

・バッチ正規化とは？

→ミニバッチ単位で入力値のデータの偏りを抑制する手法。学習を安定化し学習時間を短縮化する。また、過学習を抑制する効果もある。

・バッチ正規化の使いどころとは？

→活性化関数に値を渡す前後に、バッチ正規化処理の層を加える

◆実装演習結果キャプチャ

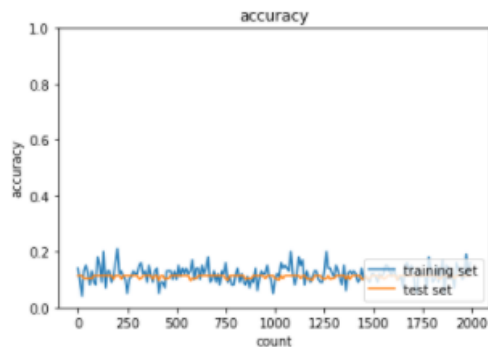
a) 活性化関数の選択

<2_2_2_vanishing_gradient_modified.ipynb>

活性化関数の選択では、sigmoid 関数では学習による正答率が向上しない場合でも、ReLU 関数により正答率向上が確認できる

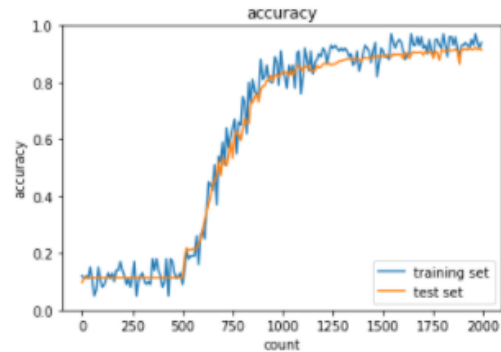
sigmoid – gauss

Generation: 2000. 正答率(トレーニング) = 0.16
: 2000. 正答率(テスト) = 0.1135



ReLU - gauss

Generation: 2000. 正答率(トレーニング) = 0.94
: 2000. 正答率(テスト) = 0.9138



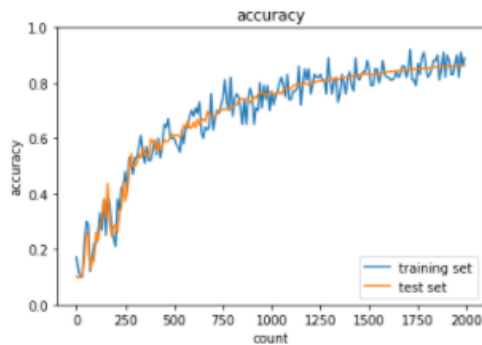
b) 重みの初期値設定

< 2_2_2_vanishing_gradient_modified.ipynb >

重みの初期値設定の変更では、Xavier では学習の進捗に合わせて徐々に正答率が向上するのに対し、HE は一気に正答率が向上することを確認できる

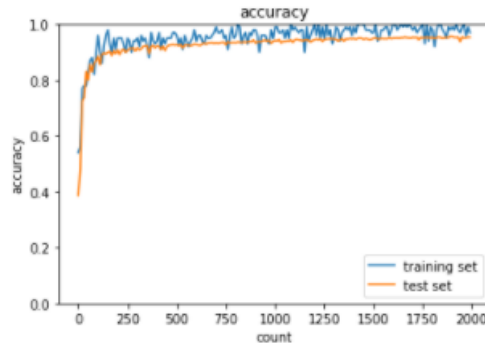
sigmoid - Xavier

Generation: 2000. 正答率(トレーニング) = 0.89
: 2000. 正答率(テスト) = 0.8637



ReLU - He

Generation: 2000. 正答率(トレーニング) = 0.97
: 2000. 正答率(テスト) = 0.9536

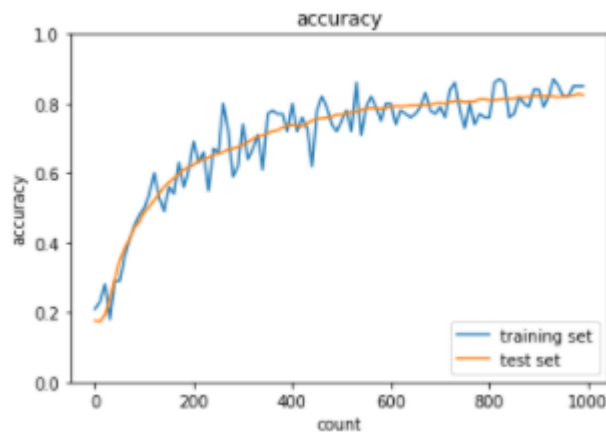


c) バッチ正規化

< 2_3_batch_normalization.ipynb >

学習データをミニバッチに分割し、各ミニバッチに対してバッチ正規化を行うことで正答率が向上することが確認できる

Generation: 1000. 正答率(トレーニング) = 0.85
 : 1000. 正答率(テスト) = 0.8249



◆確認テストなどの考察結果

確認テスト	問題	解答	考察 (コメント)
①	連鎖律の原理を使い、 $\frac{dz}{dx}$ を求めよ。 $z = t^2$ $t = x + y$	$\frac{dz}{dx} = \frac{dz}{dt} \frac{dt}{dx} = 2t \times 1$ $= 2(x + y)$	連鎖律の原理を用いて、複数の関数が合成された合成関数に対して、合成関数の微分を各関数の微分の積で求めることが出来る
②	シグモイド関数を微分した時、入力値が0の時に最大値をとる。その値として正しいものを選択肢から選べ。 (1) 0.15 (2) 0.25 (3) 0.35 (4) 0.4518	解答：(2) 0.25 図：シグモイド関数 	シグモイド関数の微分結果は以下となり、0の時に最大値、0.25となる 図：シグモイド関数微分値
③	重みの初期値に0を設定すると、どのような問題	重みを0で初期化すると正しい学習が行えない。	均一的な更新を避け、多数のノード(重み)により表現力を高めることが出来る初期値を、Xavierの初期値

	が発生するか。簡潔に説明せよ。	→すべての重みが均一に更新されるため、多数の重みをもつ意味がなくなる	や He の初期値により設定する必要がある
④	一般的に考えられるバッチ正規化の効果を 2 点挙げよ。	ミニバッチ単位で入力値のデータの偏りを抑制するため、 1. 学習を安定化し学習時間を短縮化 2. 過学習を抑制	バッチ正規化の実装は、活性化関数に値を渡す前後に、バッチ正規化処理の層を加える

<Section2: 学習率最適化手法>

◆要点まとめ

<深層学習の目的>

学習により誤差を最小にするネットワークを作成すること（誤差 $E(\mathbf{w})$ を最小化するパラメータ \mathbf{w} を発見すること）



勾配降下法を利用してパラメータを最適化

【勾配降下法】

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E, \nabla E = \frac{\partial E}{\partial \mathbf{w}} = \frac{\partial E}{\partial \mathbf{w}_1} \dots \frac{\partial E}{\partial \mathbf{w}_M}, \epsilon: \text{学習率}$$

- ・学習率が大きすぎると、演算結果が発散してしまい最適値が求まらない
- ・学習率が小さすぎると、収束するまでに時間がかかる。また、大域局所最適値に収束しづらくなる（局所解に収束する場合が発生するため）



学習率は学習効果に大きく影響する

<学習率設定方法の指針>

- ・初期の学習率を大きく設定し、徐々に学習率を小さくしていく
- ・パラメータ毎に学習率を可変にする

学習率最適化手法として以下がある。それぞれの特徴を表 7-1 に示す。また、実装内容を表 7-2 に示す。

- ・モメンタム
- ・AdaGrad
- ・RMSProp
- ・Adam

表 7-1: 学習率最適化手法の比較

	勾配降下法 (SGD)	モメンタム	AdaGrad	RMSProp	Adam

概要	誤差をパラメータで微分した値と学習率の積を減算する	誤差をパラメータで微分した値と学習率の積を減算した後、前回の重みに慣性の積を加算する	誤差をパラメータで微分した値と再定義した学習率の積を減算する	誤差をパラメータで微分した値と再定義した学習率の積を減算する	以下、2つの特性を持つアルゴリズムである ・モメンタムの、過去の勾配の指数関数的減衰平均 ・RMSProp の、過去の勾配の 2 乗の指数関数的減衰平均
メリット	—	・局所的最適解にはならず、大域的最適解となる ・谷間についてから最適値に行くまでの時間が早い	勾配の緩やかな斜面に対して、最適値に近づく	・局所的最適解にはならず、大域的最適解となる ・ハイパーパラメータの調整が必要な場合が少ない	モーメンタムと RMSProp の 2 つの特徴を合わせ持つ
課題	—	—	学習率が徐々に小さくなるため、鞍点問題が発生することがある	—	—
数式	$w^{(t+1)} = w^{(t)} - \epsilon \nabla E$ $\epsilon: \text{学習率}$	$V_t = \mu V_{t-1} - \epsilon \nabla E$ $w^{(t+1)} = w^{(t)} + V_t$ $\mu: \text{慣性}$	$h_0 = \theta$ $h_t = h_{t-1} + (\nabla E)^2$ $w^{(t+1)} = w^{(t)} - \epsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$ $\theta: \text{学習率初期値}$	$h_t = \alpha h_{t-1} + (1 - \alpha)(\nabla E)^2$ $w^{(t+1)} = w^{(t)} - \epsilon \frac{1}{\sqrt{h_t} + \theta} \nabla E$ $\theta: \text{学習率初期値}$ $\alpha: \text{前回の勾配情報を使う割合}$	$m_t = m_{t-1} + (1 - \beta_1)(\nabla E - m_{t-1})$ $v_t = v_{t-1} + (1 - \beta_2)((\nabla E)^2 - v_{t-1})$ $w^{(t+1)} = w^{(t)} - \epsilon \frac{1}{\sqrt{v_t} + \theta} m_t$ $\theta: \text{学習率初期値}$ $\beta_1, \beta_2: \text{前回の勾配情報を使う割合}$

表 7-2: 学習率最適化手法の実装比較

	コード
勾配降下法 (SGD)	<pre>network.params[key] -= learning_rate * grad[key]</pre>
モメンタム	<pre>if i == 0: v[key] = np.zeros_like(network.params[key]) v[key] = momentum * v[key] - learning_rate * grad[key] network.params[key] += v[key]</pre>
AdaGrad	<pre>if i == 0: h[key] = np.full_like(network.params[key], 1e-4) else: h[key] += np.square(grad[key]) network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[key]))</pre>
RMSProp	<pre>if i == 0: h[key] = np.zeros_like(network.params[key]) h[key] *= decay_rate h[key] += (1 - decay_rate) * np.square(grad[key]) network.params[key] -= learning_rate * grad[key] / (np.sqrt(h[key]) + 1e-7)</pre>
Adam	<pre>if i == 0: m[key] = np.zeros_like(network.params[key]) v[key] = np.zeros_like(network.params[key]) m[key] += (1 - beta1) * (grad[key] - m[key]) v[key] += (1 - beta2) * (grad[key] ** 2 - v[key]) network.params[key] -= learning_rate_t * m[key] / (np.sqrt(v[key]) + 1e-7)</pre>

※コードは「2_4_optimizer_after.ipynb」から抜粋

◆実装演習結果キャプチャ

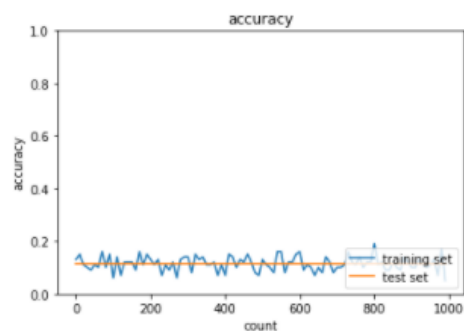
<2_4_optimizer.ipynb>

勾配降下法(SGD)に対して、モメンタム、AdaGrad、RMSProp、Adam の各学習率最適化手法を適用して正答率の変化を確認した。

モメンタム、AdaGrad では SGD に対して学習による正答率が向上しなかったが、RMSProp と Adam では正答率が向上することを確認した。なお、AdaGrad は、表 7-2 の実装内容で実装した結果である。

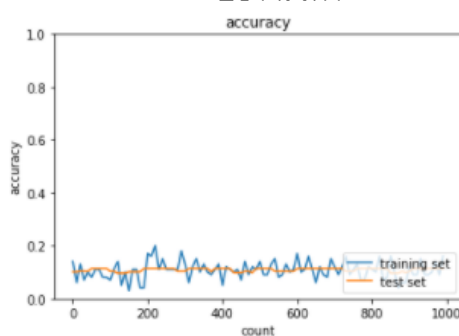
SGD

Generation: 1000. 正答率(トレーニング) = 0.05
: 1000. 正答率(テスト) = 0.1135



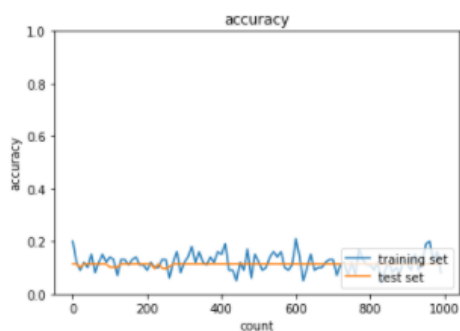
Momentum

Generation: 1000. 正答率(トレーニング) = 0.16
: 1000. 正答率(テスト) = 0.1135



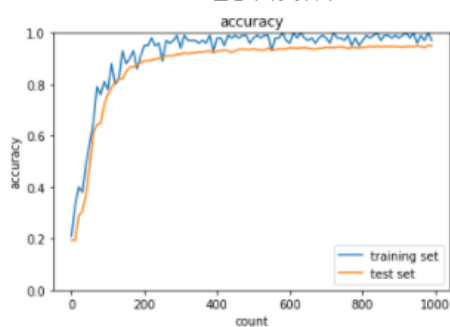
AdaGrad

Generation: 1000. 正答率(トレーニング) = 0.08
: 1000. 正答率(テスト) = 0.1135



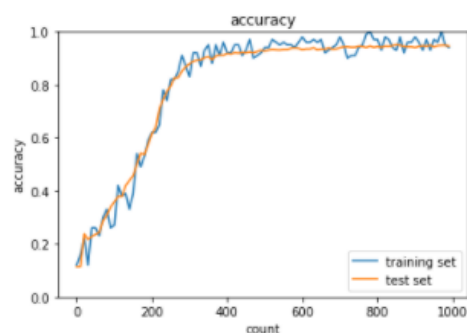
RMSprop

Generation: 1000. 正答率(トレーニング) = 0.97
: 1000. 正答率(テスト) = 0.9487



Adam

Generation: 1000. 正答率(トレーニング) = 0.94
: 1000. 正答率(テスト) = 0.9455



◆確認テストなどの考察結果

確認テスト：モメンタム・AdaGrad・RMSProp の特徴をそれぞれ簡潔に説明せよ。

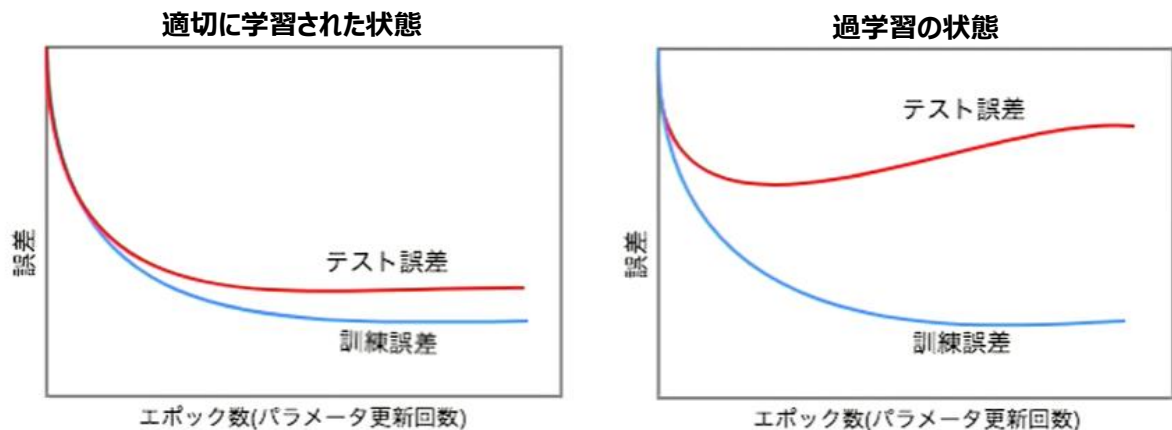
解答：表 7-1 参照

<Section3: 過学習>

◆要点まとめ

<過学習とは>

テスト誤差と訓練誤差とで学習曲線が乖離することで、特定の訓練データに対して特化して学習した結果、テストデータに対してテスト誤差が大きくなること。



<過学習の原因>

- パラメータの数が多い
- パラメータの値が適切でない
- ノードが多い etc...

↓

入力数が少ないのにニューラルネットワークが大きい場合に表現力が豊かになってしまい、ネットワークの自由度(層数、ノード数、パラメータの値 etc...)が高い場合に過学習が起こりやすい

3-1:L1 正則化、L2 正則化

正則化とは、ネットワークの自由度(層数、ノード数、パラメータの値 etc...)を制約すること

↓

正則化手法を利用して過学習を制御する

Weight decay(荷重減衰)

・過学習の原因

重みが大きい値をとることで、過学習が発生することがある

学習させていくと、重みにばらつきが発生する。重みが大きい値は、学習において重要な値として過大評価されて特定の重みに反応した過学習が起こる

・過学習の解決策

誤差に対して、正則化項を加算することで、重みを抑制する。

過学習がおこりそうな重みの大きさ以下で重みをコントロールし、

かつ、重みの大きさにばらつきを出す必要があり、重みを正則項で調整する。

<正則化手法>

誤差関数に、 p ノルム $\|x\|_p$ を加える

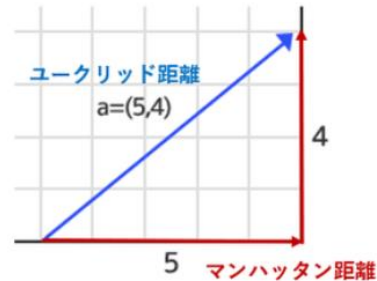
$$E_n(w) + \frac{1}{p} \lambda \|x\|_p$$

$$\|x\|_p = (|x_1|^p + \dots + |x_n|^p)^{\frac{1}{p}}$$

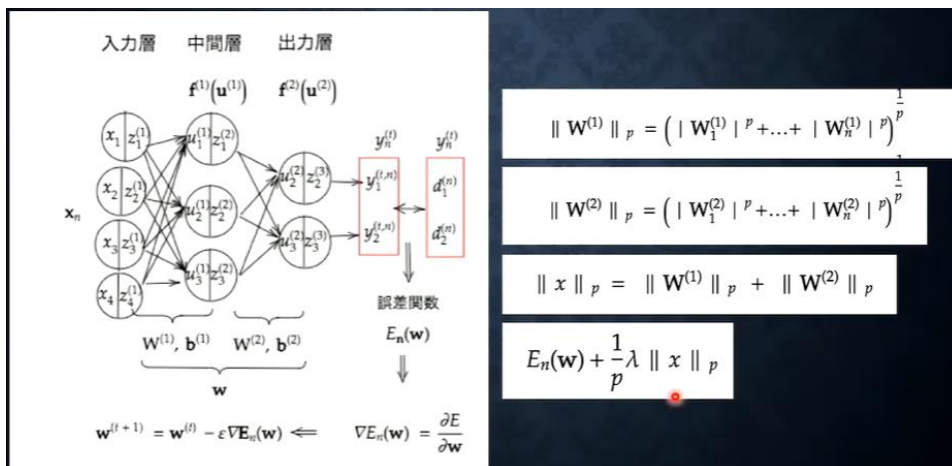


$p=1$ ($p1$ ノルム) の場合、L1 正則化と呼び、マンハッタン距離を表す

$p=2$ ($p2$ ノルム) の場合、L2 正則化と呼び、ユークリッド距離を表す



重みに対する正則化の計算：

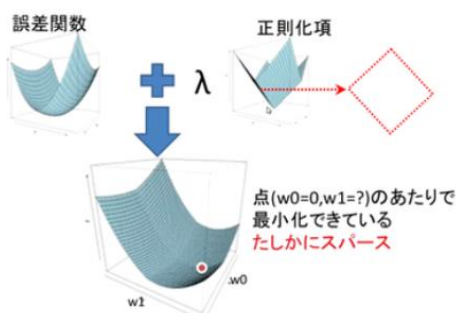


正則化項の効果：

L1 正則化では、重みのいずれかが 0 の位置で最適解になるためスパース化できる

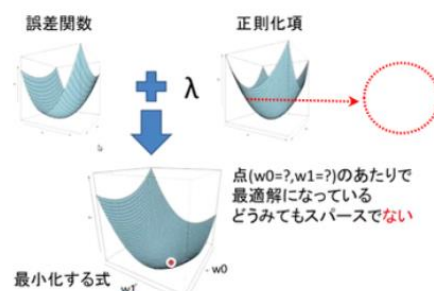
L1正則化(Lasso)

$P=1$ のときの最小化



L2正則化(Ridge)

$P=2$ のときの最小化



正則化項の実装：

1. L1 正則化項

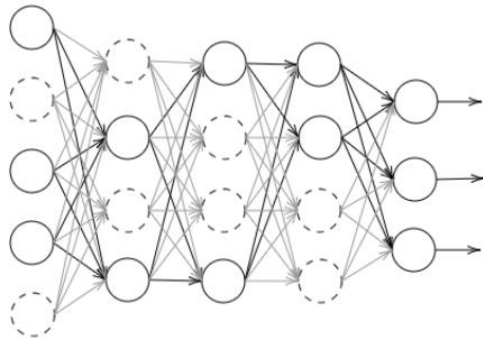
```
weight_decay += weight_decay_lambda * np.sum(np.abs(network.params['W' + str(idx)]))
```

2. L2 正則化項

```
weight_decay += 0.5 * weight_decay_lambda * np.sqrt(np.sum(network.params['W' + str(idx)] ** 2))
```

3-2:ドロップアウト

ドロップアウトとは、ランダムにノード数を削除してネットワークの自由度を制約すること



<ドロップアウトのメリット>

データ量を変化させずに異なるモデルを学習させていると解釈できる。中間層に対してドロップアウトを行うことで過学習を解消することが出来る

◆実装演習結果キャプチャ

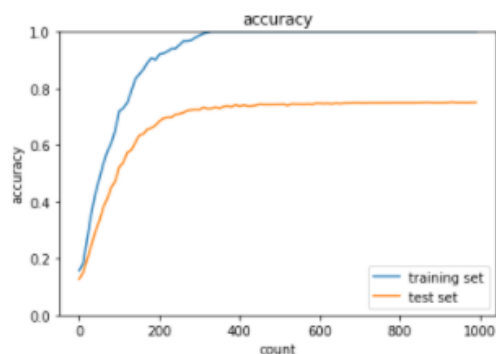
<2_4_optimizer.ipynb>

過学習が発生しているデータに対して、L2 正則化、L1 正則化と、ドロップアウトの効果を確認した。

いずれの手法も学習データに対する過学習は解消されたが、テストデータに対する正答率は向上しなかった。ドロップアウトではノード削除により学習途中の結果となっているが、ドロップアウトにL1 正則化を組み合わせて学習効率が向上した。

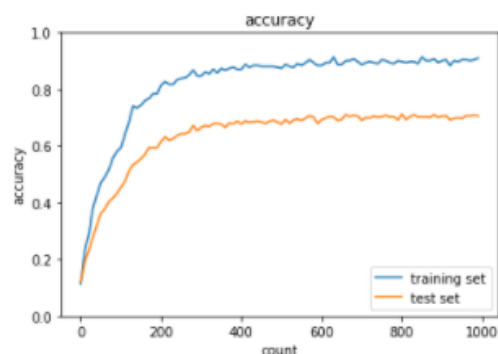
対策なし

Generation: 1000. 正答率(トレーニング) = 1.0
: 1000. 正答率(テスト) = 0.7507



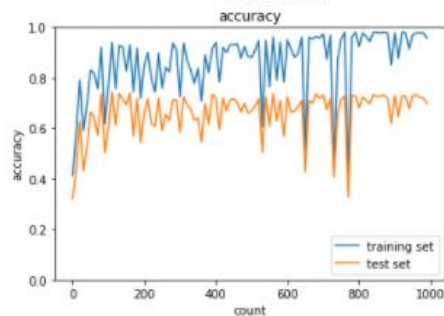
L2 正則化

Generation: 1000. 正答率(トレーニング) = 0.91
: 1000. 正答率(テスト) = 0.7056



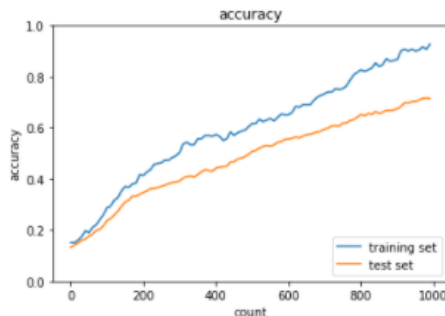
L1 正則化

Generation: 1000. 正答率(トレーニング) = 0.9566666666666667
: 1000. 正答率(テスト) = 0.6973



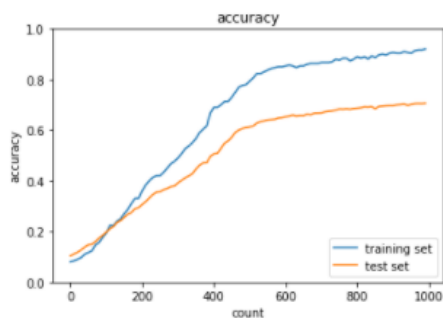
ドロップアウト

Generation: 1000. 正答率(トレーニング) = 0.9266666666666667
: 1000. 正答率(テスト) = 0.7147



L1 正則化 + ドロップアウト

Generation: 1000. 正答率(トレーニング) = 0.92
: 1000. 正答率(テスト) = 0.7063



◆確認テストなどの考察結果

確認テスト①

機械学習で使われる線形モデル(線形回帰、主成分分析・・・etc)の正則化は、モデルの重みを制限することで可能となる。前述の線形モデルの正則化手法の中にはリッジ回帰という手法があり、その特徴として正しいものを選択しなさい。

- (a)ハイパーパラメータを大きな値に設定すると、すべての重みが限りなく0に近づく
- (b)ハイパーパラメータを0に設定すると、非線形回帰となる
- (c)バイアス項についても、正則化される
- (d)リッジ回帰の場合、隠れ層に対して正則項を加える

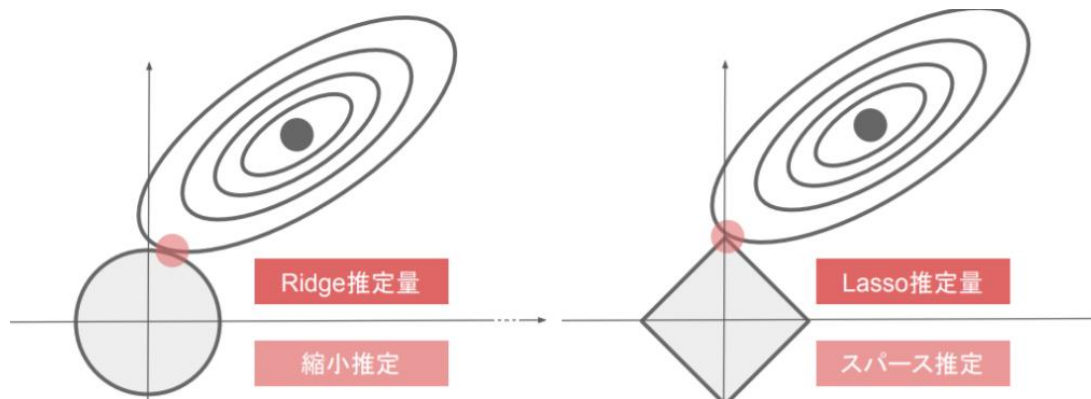
<解答>

(a)

ハイパーパラメータを大きな値にすると正則化項の影響が大きくなり、正則化項の最小値となる0に近づく

確認テスト②

下図について、L1 正則化を表しているグラフはどちらか答えよ。



<解答>

Lasso 推定量

L1 正則化では正則化項の形がひし形になり、いずれかの重みが0の点が最適解となる

<Section4: 畳み込みニューラルネットワークの概念>

◆要点まとめ

<全結合層で画像を学習した際の課題>

画像の場合、縦、横、チャンネルの3次元データだが、1次元のデータとして処理する。

→全結合層ではRGBの各チャンネル間の関連性が、学習に反映されない。



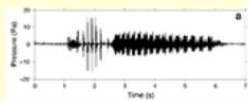
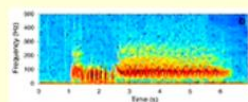
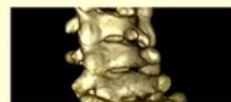



そこで考え出されたのが、CNN (Convolutional Neural Network) である。

<CNN (Convolutional Neural Network) >

CNN では、図 9-1 のような次元間で繋がりのあるデータを扱うことができる。

- 単チャンネルでは、主に強度などのデータとなり
 - 1次元：音声データの振幅の強度
 - 2次元：画像データのフーリエ変換後のスペクトラムの強度
 - 3次元：CT画像のX線の透過強度
 などである。
- 複数チャンネルでは、複数の値のデータとなり
 - 1次元：アニメスケルトンの時系列に変化する要素
 - 2次元：画像データの時系列に変化するRGB信号
 - 3次元：動画データの時系列に変化するフレーム
 などである。

図 9-1: CNN で扱うデータ

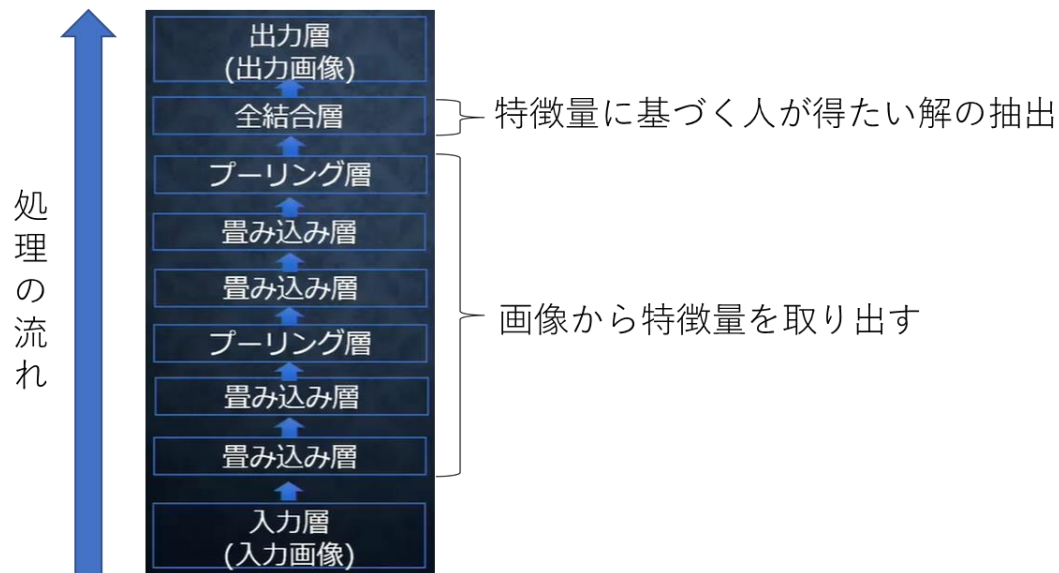
	1次元	2次元	3次元
単一チャンネル	音声 [時刻, 強度] 	フーリエ変換した音声 [時刻, 周波数, 強度] 	CTスキャン画像 [x, y, z, 強度] 
複数チャンネル	アニメのスケルトン [時刻, (腕の値, 膝の値 …)] 	カラー画像 [x, y, (R, G, B)] 	動画 [時刻, x, y, (R, G, B)] 

<CNN の構造図(例)>

CNN の構造図例とCNN 内部での役割イメージを図 9-2 に示す。

最終的に人が得たい解を求める処理は全結合層のニューラルネットワーク部分で行われるが、前半部分で大量の次元間で繋がりのあるデータから特徴量を抽出してニューラルネットワーク部分へ渡すイメージとなる。

図 9-2: CNN の構造図(例)

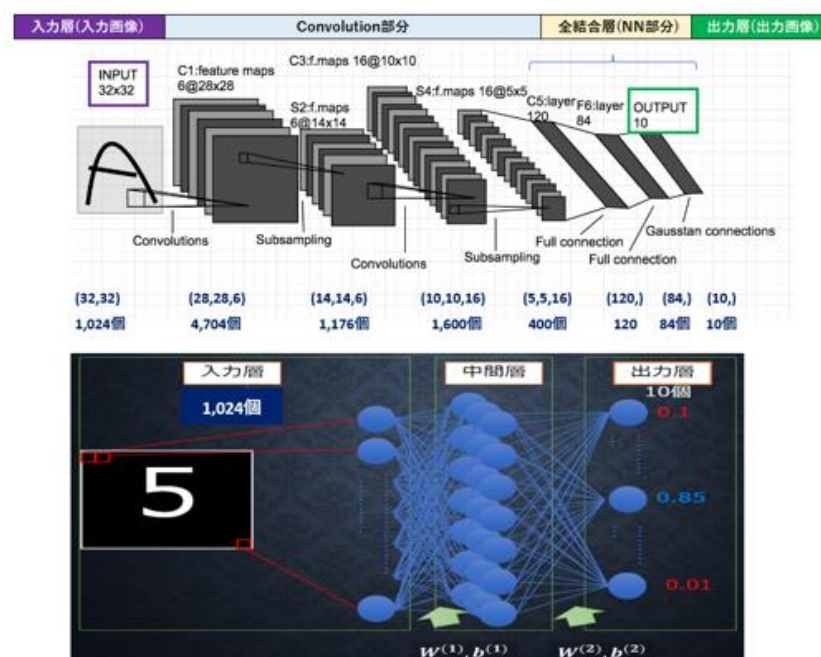


<LeNet>

LeNet は、畳み込みニューラルネットワーク (Convolutional Neural Network, CNN) の機構であり、1989 年にヤン・ルカン (Yann LeCun) らによって提案された。LeNet という語は、一般に、単純な畳み込みニューラルネットワークである LeNet-5 を指す。畳み込みニューラルネットワークはフィードフォワード・ニューラルネットワークの一種であり、人工ニューロンが周囲の細胞の一部をカバー範囲内として応答することができ、大規模な画像処理に適している。(Wikipedia より)

LeNet の構成を図 9-3 に示す。

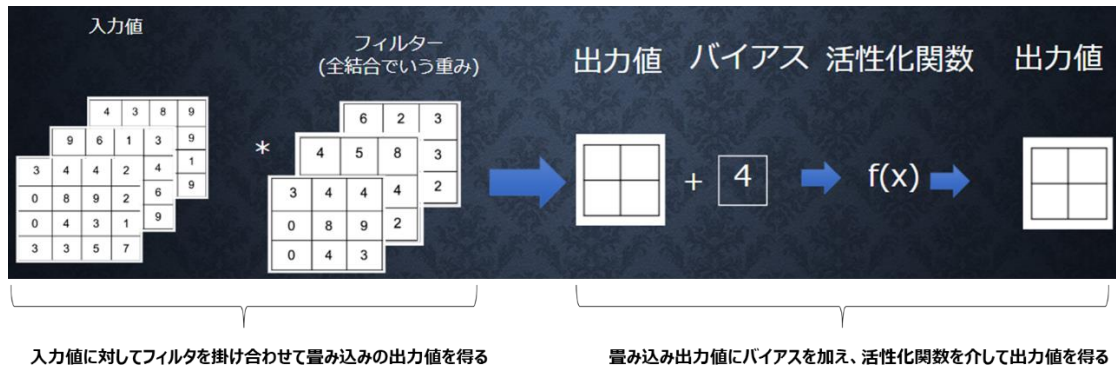
図 9-3: LeNet



<畳み込み層の全体像>

畳み込み層では、画像の場合、縦、横、チャンネルの 3 次元データをそのまま学習し、次に伝えることができる。3 次元の空間情報も学習できるような層が畳み込み層である。

図 9-4: 畳み込み層



以下では、畳み込み層の演算要素を説明する。

a. フィルター

入力画像に対してフィルタを掛け合わせることで、畳み込みの出力値を得る。

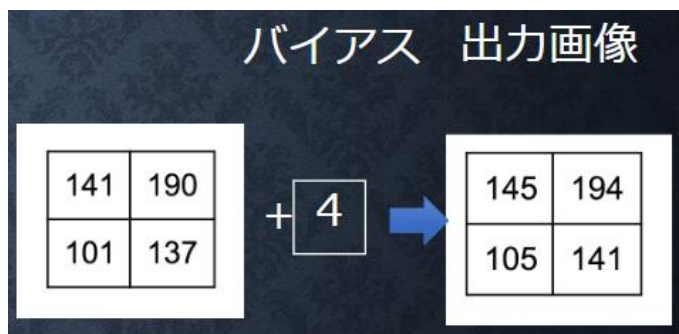


フィルター演算 ※赤枠部分

$$3 \times 3 + 4 \times 1 + 4 \times 2 + 0 \times 8 + 8 \times 7 + 9 \times 5 + 0 \times 5 + 4 \times 4 + 3 \times 1 = 141$$

b. バイアス

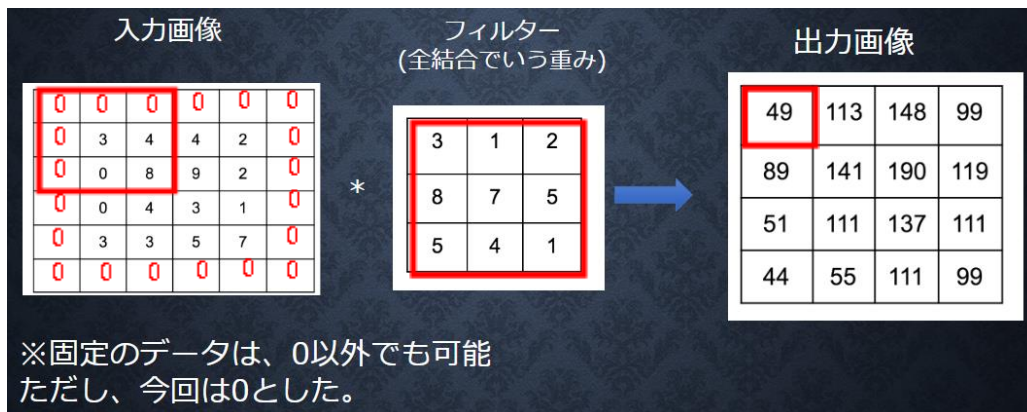
バイアスは、畳み込み出力画素全体に加算する値である。



c. パディング

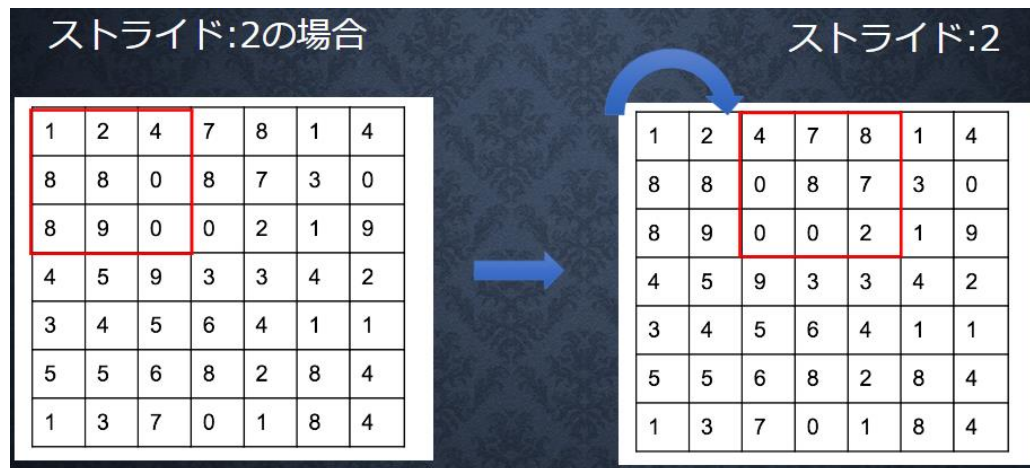
フィルターを掛け合わせることで出力画像サイズが得られるように、入力画像の上下左右の画素を埋めて画像サイズを拡張すること。パディングデータは0などの固定値でも良いし、画素値が連続になるように、パディング対象画素の近傍の画素値でも良い。

以下の例では、赤字の画素に対して0でパディングしている。



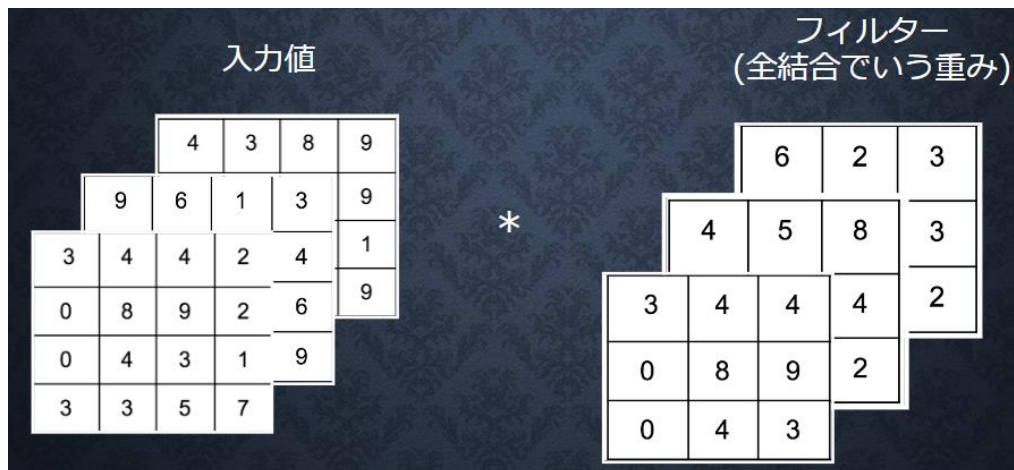
d. スライド

畳み込み処理で入力画像に対してフィルターを適用する間隔のこと、以下の様に、ストライド: 2 の場合は、フィルターを横に2画素ずつずらしながら畳み込み演算を行う。ストライドには水平方向と垂直方向がある。



e. チャンネル

以下の様に RGB 画像であれば3チャンネルの画像となる。畳み込み演算は、各チャンネルの入力画像に対してフィルターを掛け合わせて畳み込み演算を行う。



<畳み込み演算の実装テクニック>

畳み込み演算の前に、入力画像をフィルターと掛け合わせる範囲の行列に並び変える。

こうすることで、並び替えた入力画像の行列とフィルター行列の行列積を高速に求めることが出来る
img2col クラスで実装される

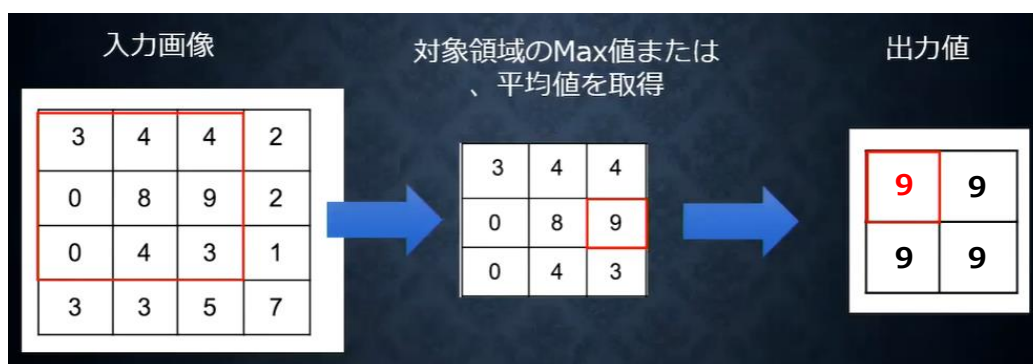
<プーリング層>

入力画像に対して対象領域の最大値、または、平均値を取得する処理。これにより、入力画像の中から重要だと思われる情報を抜き出して処理対象のデータサイズを小さくする。プーリング層は畳み込み層の間に挟むことが多く、画像の解像度(サイズ)をシュリンクする効果がある。

対象領域の最大値を取得する処理を Max Pooling、平均値を取得する処理を Average Pooling と呼ぶ。

図 9-5 では、3x3 の領域に対して Max プーリングで演算した例を示す。

図 9-5: プーリング層



◆実装演習結果キャプチャ

a) 畳み込み層

<2_6_simple_convolution_network_after.ipynb>

・im2colの確認

フィルタ高さ、幅が 3、ストライドが 1、パディングが 0 の場合の、以下 2 つの入力画像を、im2col で変換した結果を示す。1 つの画像あたり 4×9 の行列に変換されている。

```
===== input_data =====
[[[85. 83. 42. 59.]
  [95. 51. 62. 18.]
  [12.  6.  7. 88.]
  [ 0. 45. 37. 26.]]

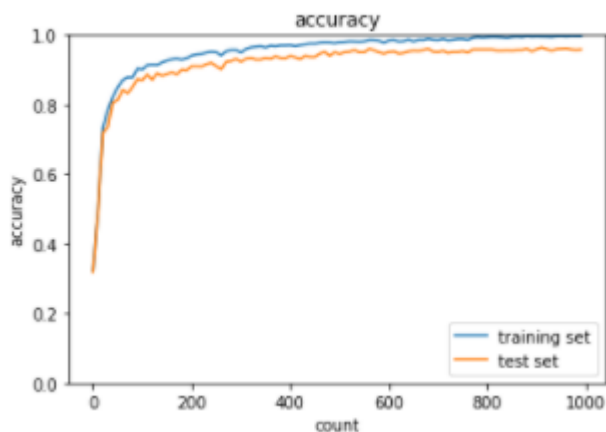
 [[58. 25. 90. 60.]
  [33. 32. 96. 82.]
  [96. 24. 21. 18.]
  [65. 63. 65. 49.]]]]
=====
===== col =====
[[85. 83. 42. 95. 51. 62. 12.  6.  7.]
 [83. 42. 59. 51. 62. 18.  6.  7. 88.]
 [95. 51. 62. 12.  6.  7.  0. 45. 37.]
 [51. 62. 18.  6.  7. 88. 45. 37. 26.]
 [58. 25. 90. 33. 32. 96. 96. 24. 21.]
 [25. 90. 60. 32. 96. 82. 24. 21. 18.]
 [33. 32. 96. 96. 24. 21. 65. 63. 65.]
 [32. 96. 82. 24. 21. 18. 63. 65. 49.]]
=====
```

・CNNの確認

load_mnist()で取得した、訓練画像、訓練ラベルにより CNN で学習を 1000 回行い、学習時の訓練データとテストデータのそれぞれの正答率を確認した。以下に結果を示す。

実行時間として 13 分かかったが訓練画像 5000 枚に対して十分な正答率が出ているため、用途に応じて学習回数を減らしたり、訓練画像を減らして実行時間を短くしても良いと思われる。

Generation: 1000. 正答率(トレーニング) = 0.996
: 1000. 正答率(テスト) = 0.958



◆確認テストなどの考察結果

<問題>

サイズ 6×6 の入力画像を、サイズ 2×2 のフィルターで畳み込んだ時の
出力画像のサイズを答えよ。なおストライドとパディングは 1 とする。
(3 分)

<解答>

出力画像のサイズ 7×7

$$\text{高さ } h = \{(6 + 2 \cdot 1 - 2) \div 1\} + 1 = 7$$

$$\text{幅 } w = \{(6 + 2 \cdot 1 - 2) \div 1\} + 1 = 7$$

※以下の畳み込み後の出力画像サイズを求める公式を使用する

畳み込み後の出力画像サイズを求める公式

$$O_H = \frac{\text{画像の高さ} + 2 \times \text{パディング高さ} - \text{フィルター高さ}}{\text{ストライド幅}} + 1$$

$$O_W = \frac{\text{画像の幅} + 2 \times \text{パディング幅} - \text{フィルター幅}}{\text{ストライド幅}} + 1$$

<Section5: 最新の CNN>

◆要点まとめ

AlexNet

AlexNet は畳み込みニューラル ネットワーク (CNN) のアーキテクチャの名前であり、Alex Krizhevsky が博士課程の指導教官である Ilya Sutskever および、ジェフリー・ヒントンと共同で設計した。

AlexNet は、2012 年 9 月 30 日に開催された ILSVRC 2012[3] に参加した。AlexNet はエラー率 15.3%で優勝し、次点よりも 10.8%以上低かった。

この論文の主な内容は、モデルの深さが高性能には不可欠であるというもので、計算コストは高くなるものの、GPU を用いて学習することで実現した。(Wikipedia より)

これにより、人が特徴量を設計しなくても、十分なデータさえ存在すれば、機械自身が特徴量を見つけ出すことが AlexNet によって示された。

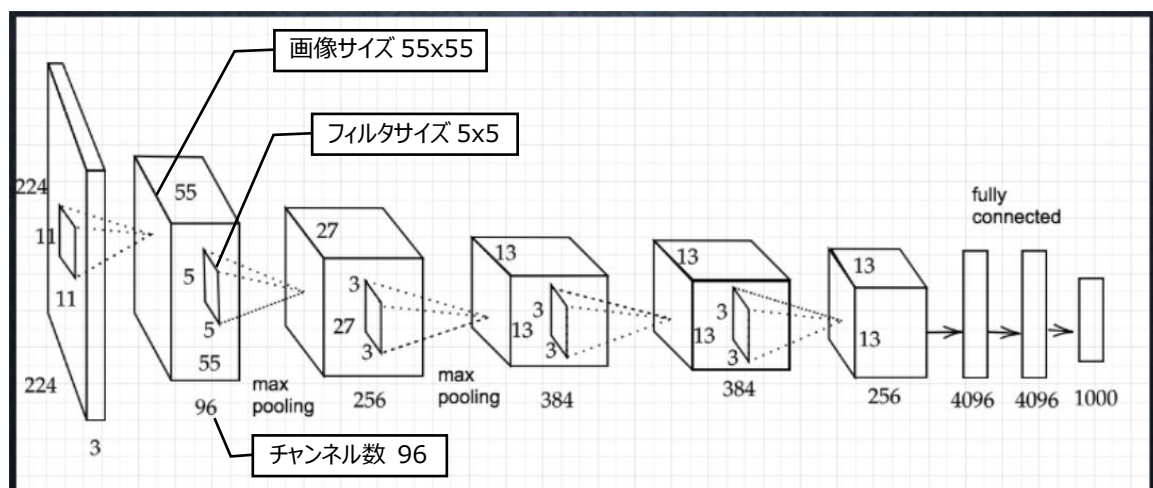
図 10 に、AlexNet の構成図を示す。

5 層の畳み込み層およびプーリング層と、3 層の全結合層で構成される。畳み込み層の出力画像 (13x13,256 チャンネル)から全結合層への 4096 個の入力データへの変換は、Fratten という処理でサンプリングして行う。

Fratten では、 $13 \times 13 \times 256 = 43264$ のデータに変換する。他にも Global Max Pooling や Global Average Pooling の変換手法がある。前者は各チャンネルから最大値を出力する処理であり、後者は各チャンネルの平均値を出力する処理である。

また、過学習を防ぐ施策として、全結合層の出力にドロップアウトを使用している。

図 10: AlexNet 構成図



◆実装演習結果キャプチャ

実装演習対象のコード説明とコード提供なしのため省略

◆確認テストなどの考察結果

確認テストの出題なしのため省略