

## 深層学習 day1 レポート

### 1 入力層～中間層

#### 1.1 要点のまとめ

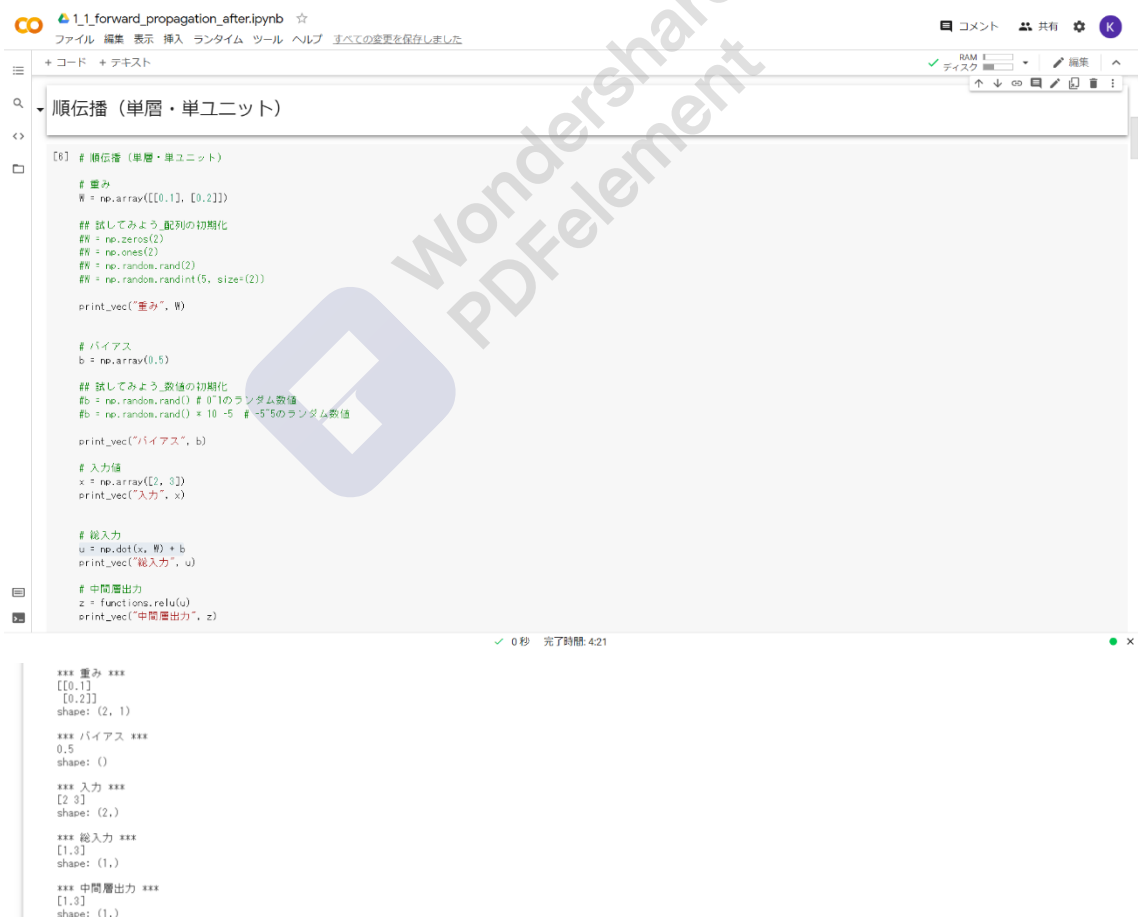
- ・ 入力層は入力  $x_i$ 、バイアス  $b$  から構成される。(i は入力層のインデックス)
- ・ 重みは  $w_i$ 、で表し各入力に対して掛ける。
- ・ 入力が  $n$  個ある場合、総入力  $u$  は下記の式となる。

$$u = w_1x_1 + w_2x_2 + w_3x_3 \cdots + w_nx_n$$

- ・ 中間層は総入力に対し活性化関数  $f$  を実行しその結果  $z$  を出力層に渡す。

$$z = f(u)$$

#### 1.2 演習結果



```
1_1_forward_propagation_after.ipynb ☆
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました

+ コード + テキスト
RAM ディスク
コメント 共有 設定 K
↑ ↓ ⌂ 🔍 📄 ⋮

順伝播 (単層・単ユニット)

[0] # 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
w0 = np.zeros(2)
w1 = np.ones(2)
w2 = np.random.rand(2)
w3 = np.random.randint(5, size=(2))

print_vec("重み", w)

# バイアス
b = np.array(0.5)

## 試してみよう_数値の初期化
b0 = np.random.rand() # 0~1のランダム数値
b1 = np.random.rand() * 10 -5 # -5~5のランダム数値

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)

*** 重み ***
[[0.1]
 [0.2]]
shape: (2, 1)

*** バイアス ***
0.5
shape: ()

*** 入力 ***
[2 3]
shape: (2, )

*** 総入力 ***
[1.3]
shape: (1, )

*** 中間層出力 ***
[1.3]
shape: (1, )
```



```
1_1_forward_propagation_after.ipynb ☆
ファイル 編集 表示 挿入 ランタイム ツール ヘルプ すべての変更を保存しました
+ コード + テキスト
RAM
ディスク
編集
順伝播 (3層・複数ユニット)

[7] # 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 = []

    input_layer_size = 3
    hidden_layer_size_1=10
    hidden_layer_size_2=5
    output_layer_size = 4

    #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.random.rand(input_layer_size, hidden_layer_size_1)
    network['W2'] = np.random.rand(hidden_layer_size_1, hidden_layer_size_2)
    network['W3'] = np.random.rand(hidden_layer_size_2, output_layer_size)

    network['b1'] = np.random.rand(hidden_layer_size_1)
    network['b2'] = np.random.rand(hidden_layer_size_2)
    network['b3'] = np.random.rand(output_layer_size)

    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):

0秒 完了時間: 4.46
```

```
# 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("中間層出力2", z2)
    print_vec("総入力2", u2)
    print_vec("出力", y)
    print("出力合計: " + str(np.sum(y)))

    return y, z1, z2

# 入力値
x = np.array([1., 2., 4.])
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. 4.]
shape: (3, )

##### ネットワークの初期化 #####
### 重み ###
[[[0.10721256 0.85624021 0.06875953 0.47311676 0.46620253 0.81668304
  0.1847623 0.81550938 0.35659561 0.51101024]
 [0.44025978 0.80560288 0.38025701 0.12710752 0.14796809 0.25369637
  0.35068679 0.7508298 0.8914387 0.06515187]
 [0.61096422 0.52823591 0.5607337 0.53451291 0.1480483 0.56544648
  0.02711842 0.48418062 0.87322964 0.67728006]]
 shape: (3, 10)

### 偏り ###
[[[5.45470129e-01 9.27481375e-01 4.60732670e-01 7.31200890e-01
  1.36200210e-01]
 [4.44719603e-01 4.46259400e-01 5.38043952e-02 9.32627238e-01
  8.9493399e-01]
 [4.94124423e-01 6.96531409e-01 7.96250410e-01 6.98024804e-01
  2.78079186e-01]
 [6.08448886e-01 2.54544698e-04 5.84855603e-01 6.44085499e-01
  5.86454637e-02]
 [8.86372512e-01 7.40989556e-01 9.40742319e-01 1.26408447e-01
  2.31705800e-01]
 [4.67536793e-01 6.6220212e-01 8.42223490e-01 4.40074204e-01
  5.26325617e-01]
 [6.12361518e-01 2.63807180e-01 3.49631985e-01 6.46054172e-02
  6.27606387e-01]
 [9.35553214e-01 9.59610222e-02 5.16259823e-01 7.59600430e-01
  3.40550017e-01]
 [6.18014038e-02 8.30270417e-01 3.79102590e-01 8.59807210e-01
  6.38290644e-01]
 [4.80045676e-01 2.28043291e-01 6.44894995e-03 9.72667131e-01
  6.21624081e-01]]
 shape: (10, 5)
```

```
### 重み2 ###
[[[0.06975493 0.24249043 0.34330181 0.34862664]
 [0.85269695 0.34871588 0.59239805 0.1487325 ]
 [0.52772088 0.30128876 0.70056375 0.18675774]
 [0.74207014 0.29568607 0.81965398 0.06751455]
 [0.94553999 0.74785947 0.14686288 0.95960233]]
 shape: (5, 4)

### バイアス1 ###
[0.83382091 0.83456574 0.00213178 0.67905881 0.03484852 0.07867223
 0.8478942 0.53383581 0.19236255 0.42897255]
 shape: (10, )

### バイアス2 ###
[0.08561206 0.81640375 0.89460332 0.16862552 0.51388874]
 shape: (5, )

### バイアス3 ###
[0.71977224 0.21534389 0.67194513 0.62618296]
 shape: (4, )

##### 順伝播開始 #####
### 入力1 ###
[4.365409 5.01496155 3.09434011 3.54444227 1.38917844 3.68454392
 1.84250373 4.78802724 6.02475414 3.77940678]
 shape: (10, )

### 中間層出力1 ###
[4.365409 5.01496155 3.09434011 3.54444227 1.38917844 3.68454392
 1.84250373 4.78802724 6.02475414 3.77940678]
 shape: (10, )

### 中間層出力2 ###
[19.26048265 19.52546287 17.49929164 26.88168956 17.93419574]
 shape: (5, )

### 総入力2 ###
[19.26048265 19.52546287 17.49929164 26.88168956 17.93419574]
 shape: (5, )

### 出力 ###
[64.85314171 38.33527178 55.77439879 32.53770151]
 shape: (4, )

出力合計: 191.50051379045777
```

### 1.3 考察

- ・Numpy関数と行列計算は相性が良い。
- ・行列積がライブラリとして用意されているため、配列計算でプログラミングをするよりかなり省力化される。
- ・中間層のノード数を増やすとネットワークは複雑に見えるが、一つ一つの処理は、意外と単純な計算であることが分かった。

## 2 活性化関数

### 2.1 要点のまとめ

- ・ニューラルネットワークにおいて、次の層への出力の大きさを決める非線形の関数のこと。
- ・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として、ReLU 関数、シグモイド（ロジスティック）関数、ステップ関数等が使用される。
- ・出力層用の活性化関数として、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）が使用される。
- ・いずれの活性化関数も 0~1 の間の値となる。

### 2.2 演習結果

```
# 順伝播 (単層・複数ユニット)

# 重み
W = np.array([
    [0.1, 0.2, 0.3, 0],
    [0.2, 0.3, 0.4, 0.5],
    [0.3, 0.4, 0.5, 1]
])

## 試してみよう_配列の初期化
## = np.zeros((4,3))
## = np.ones((4,3))
## = np.random.rand(4,3)
## = 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(W, x) + b
print_vec("総入力", u)

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

```
*** 重み ***
[[0.1 0.2 0.3 0.]
 [0.2 0.3 0.4 0.5]
 [0.3 0.4 0.5 1.]
 shape: (3, 4)

*** バイアス ***
[0.1 0.2 0.3]
 shape: (3,)

*** 入力 ***
[1. 5. 2. -1.]
 shape: (4,)

*** 総入力 ***
[1.8 2.2 2.6]
 shape: (3,)

*** 中間層出力 ***
[0.85814894 0.90024951 0.93086158]
 shape: (3,)
```

## 2.3 考察

- ・シグモイド関数等をプログラムで使用する場合は、あらかじめ関数として定義しておくといよい。
- ・活性化関数は、総入力値の値を 0~1 で標準化するようなものと理解した。
- ・どの活性化関数を使用するかについては正解が無く、実装するうえでは試行錯誤が必要
- ・出力層用の活性化関数として、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）が使用される。
- ・

## 3 出力層

### 3.1 要点のまとめ

- ・出力層の中間層との違いとして、中間層はしきい値の前後で信号の強弱をつける。出力層は信号の大きさをそのままに変換する。
- ・分類問題の場合、出力層の出力は 0~1 となり総和は 1 となる。
- ・誤差関数として、2 乗誤差関数、交差エントロピー関数が使用される。

### 3.2 演習結果

```
# 0 図解
# 2-3-2ネットワーク

# ! 試してみよう_ノードの構成を 3-5-4 に変更してみよう

# 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 #####")

    input_layer_size = 3
    hidden_layer_size = 50
    output_layer_size = 2

    # 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.random.rand(input_layer_size, hidden_layer_size)
    network['W2'] = np.random.rand(hidden_layer_size, output_layer_size)

    network['b1'] = np.random.rand(hidden_layer_size)
    network['b2'] = np.random.rand(output_layer_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

# プロセスを作成
def forward(network, x):
    print("##### 順伝達開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
```

```
W1, W2 = network['W1'], network['W2']
b1, b2 = network['b1'], network['b2']
# 隠れ層の総入力
u1 = np.dot(x, W1) + b1
# 隠れ層の総出力
z1 = functions.relu(u1)
# 出力層の総入力
u2 = np.dot(z1, W2) + b2
# 出力層の総出力
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 y, z1

# 入力値
x = np.array([1., 2., 3.])
network = init_network()
y, z1 = forward(network, x)
# 目標出力
d = np.array([2., 4.])
# 誤差
loss = functions.mean_squared_error(d, y)

## 表示
print("\n#### 結果表示 ####")
print_vec("中間層出力", z1)
print_vec("出力", y)
print_vec("訓練データ", d)
print_vec("二乗誤差", loss)
```

```
#### ネットワークの初期化 ####
```

```
*** 重み ***
```

```
[[[7.22066227e-01 7.06602756e-01 4.71306995e-01 1.81744223e-02
 6.46332310e-01 6.81840601e-01 1.36052710e-01 4.91612847e-01
 7.27341682e-01 1.95910835e-01 3.62274348e-01 8.67570134e-01
 4.91338054e-01 3.17405027e-01 2.06545665e-01 3.50375706e-01
 7.27379993e-01 3.79821326e-01 2.05579018e-01 8.19712028e-01
 6.06783783e-01 3.01820132e-02 7.11666386e-01 1.99354774e-01
 8.48006831e-01 5.34632074e-01 8.96634448e-01 5.20396898e-01
 8.94103368e-02 6.17052905e-01 3.63337184e-01 8.93159450e-02
 7.94987573e-01 2.79002026e-01 3.58047023e-01 5.94539333e-01
 1.79298585e-02 4.86941428e-01 5.07676242e-01 7.98954415e-01
 4.74108774e-01 6.73016436e-01 1.74801022e-01 3.61060966e-02
 5.28832539e-01 7.98082134e-01 3.12285056e-01 8.73808013e-01
 6.34096627e-01 2.70152967e-01]
 [9.30565633e-01 5.11552449e-01 4.58132965e-01 4.61373563e-01
 7.06470211e-01 3.84388506e-01 8.33579272e-01 5.87630256e-01
 1.42387546e-01 4.72077127e-01 6.69440408e-01 1.99098781e-01
 2.21426641e-01 7.88404720e-01 9.31433397e-01 7.73378218e-01
 5.36126551e-01 4.94151666e-01 9.94162613e-01 8.37363571e-01
 8.16662031e-01 5.10125671e-01 3.32188712e-02 8.64771495e-01
 1.09284369e-01 9.41043091e-01 9.66513172e-01 4.76333597e-01
 5.33936722e-01 2.11092751e-01 4.04756179e-01 8.39885613e-01
 7.15938466e-01 7.24283204e-01 6.35478425e-01 1.05147122e-01
 9.19921695e-01 5.92128871e-01 5.59299150e-01 4.02953017e-01
 6.98656316e-02 3.34251738e-02 1.73654141e-01 4.71352255e-01
 8.33073357e-02 3.97222756e-01 2.99456895e-04 2.08978021e-01
 2.16268681e-01 1.62069774e-01]
 [9.12046814e-01 6.36886411e-01 1.85376801e-01 3.94731113e-01
 4.95817447e-02 7.75354850e-01 3.73521443e-01 8.30692618e-01
 1.20721411e-02 9.25059122e-01 2.11396571e-01 1.43687918e-02
 2.71216478e-01 1.88707851e-01 7.17427603e-01 7.15407223e-01
 7.55041220e-01 5.52216292e-01 3.06557445e-01 5.72337860e-01
 4.80070719e-01 8.44181045e-01 3.38244921e-01 6.32523226e-01
 5.14170959e-01 6.93654646e-01 4.59803487e-01 7.52066545e-01
 5.43709964e-01 8.56614212e-01 4.50109482e-01 7.55973964e-03
 2.04568544e-01 3.89377303e-01 9.17569744e-01 5.01282028e-01
 2.66720193e-01 4.48491462e-01 3.23565890e-01 4.95045973e-01
 3.74185734e-01 6.89809078e-01 4.86278264e-01 2.57140898e-01
 3.04308416e-01 2.52165161e-02 5.19633072e-01 7.47137668e-01
 2.66624500e-01 7.01911161e-01]]]
shape: (3, 50)
```



```
*** 重み2 ***
[[0.2728002 0.80742372]
[0.23538079 0.30342153]
[0.14451651 0.07351782]
[0.5498454 0.32910514]
[0.08478737 0.46859248]
[0.61895274 0.67374897]
[0.12133661 0.80825035]
[0.99956153 0.63879734]
[0.68947734 0.8053326 ]
[0.2847543 0.57113759]
[0.62049984 0.37638017]
[0.51470528 0.88353641]
[0.98051711 0.29951014]
[0.93066751 0.34689244]
[0.4381702 0.85346736]
[0.0342808 0.64549362]
[0.95581099 0.94180392]
[0.56648106 0.24684644]
[0.38170482 0.36035703]
[0.01695135 0.352673 ]
[0.95652406 0.23675299]
[0.76754325 0.61774806]
[0.86241 0.08688665]
[0.74210053 0.97498307]
[0.4677116 0.19868822]
[0.63493378 0.62859056]
[0.39973011 0.6357975 ]
[0.40343134 0.59838438]
[0.41393148 0.47767288]
[0.77071488 0.37281025]
[0.95326558 0.17630315]
[0.36098323 0.5105498 ]
[0.45635936 0.99408379]
[0.19780154 0.3678553 ]
[0.13104947 0.55747778]
[0.38361405 0.19475171]
[0.42338365 0.25719781]
[0.25241832 0.0312506 ]
[0.57381512 0.82500591]
[0.22205616 0.81234797]
[0.80654993 0.27360237]
[0.49586751 0.83912303]
[0.51546309 0.05291325]
[0.89805602 0.49869876]
[0.38339234 0.32688226]
```

```
*** バイアス1 ***
[0.58820873 0.4546297 0.39362885 0.41189257 0.99518311 0.18999344
0.10268292 0.50380514 0.38259208 0.11806473 0.88655455 0.55125607
0.97678881 0.69684441 0.120085 0.17660906 0.23616231 0.11472737
0.12072791 0.04304302 0.40079171 0.99338668 0.69264424 0.15553174
0.18498302 0.93858607 0.36496478 0.8944798 0.97909802 0.68385575
0.85655147 0.39810031 0.84536623 0.00889486 0.64937749 0.37817884
0.7047376 0.38706863 0.61486269 0.34488808 0.05550364 0.6272412
0.26480062 0.06161787 0.34313637 0.01278688 0.37950185 0.97722225
0.15525681 0.42479483]
shape: (50,)
```

```
*** バイアス2 ***
[0.23588051 0.05351547]
shape: (2,)
```

```
##### 順伝播開始 #####
```

```
*** 総入力1 ***
[5.90754466 4.09499659 2.33733217 2.53680746 3.20320108 3.9666756
3.0264585 4.66275636 1.43092527 4.03330718 3.20189943 1.86013014
2.72462959 3.75718243 4.34178026 4.21996287 4.30091907 3.13950091
3.23430448 4.25449577 4.08111171 4.57636317 2.48548313 3.98199918
2.80407147 5.43626827 4.59403604 4.62374352 3.76750969 4.24303679
3.37972946 2.1910667 3.68593637 2.9045952 5.03109059 2.6888583
3.36267125 3.40374218 3.2118349 3.43488645 1.79190088 3.43653522
2.24574471 1.81185117 1.95150883 1.68096408 2.25128503 4.5103993
2.0217643 3.12482083]
shape: (50,)
```

```
*** 中間層出力1 ***
[5.90754466 4.09499659 2.33733217 2.53680746 3.20320108 3.9666756
3.0264585 4.66275636 1.43092527 4.03330718 3.20189943 1.86013014
2.72462959 3.75718243 4.34178026 4.21996287 4.30091907 3.13950091
3.23430448 4.25449577 4.08111171 4.57636317 2.48548313 3.98199918
2.80407147 5.43626827 4.59403604 4.62374352 3.76750969 4.24303679
3.37972946 2.1910667 3.68593637 2.9045952 5.03109059 2.6888583
3.36267125 3.40374218 3.2118349 3.43488645 1.79190088 3.43653522
2.24574471 1.81185117 1.95150883 1.68096408 2.25128503 4.5103993
2.0217643 3.12482083]
shape: (50,)
```

```
.....
```

```

*** 総入力2 ***
[86.82435722 87.49835345]
shape: (2,)

*** 出力1 ***
[86.82435722 87.49835345]
shape: (2,)

出力合計: 174.32271066994107

##### 結果表示 #####
*** 中間層出力 ***
[5.90754486 4.09499659 2.33733217 2.53680746 3.20320108 3.9666756
 3.0264585 4.66275636 1.43092527 4.03330718 3.20189943 1.86013014
 2.72462959 3.75718243 4.34178026 4.21986287 4.30091907 3.13950091
 3.23430448 4.25449577 4.08111171 4.57636317 2.48548313 3.98199918
 2.80407147 5.43626827 4.59403604 4.62374352 3.76750969 4.24303679
 3.37972948 2.1910667 3.68593637 2.9045952 5.03109059 2.6868563
 3.36267125 3.40374218 3.2118349 3.43488645 1.79190088 3.43653522
 2.24574471 1.81185117 1.95150883 1.68096408 2.25128503 4.5103993
 2.0217643 3.12482083]
shape: (50,)

*** 出力 ***
[86.82435722 87.49835345]
shape: (2,)

*** 訓練データ ***
[2. 4.]
shape: (2,)

*** 二乗誤差 ***
3541.7886518588674
shape: ()

```

### 3.3 考察

- ・出力層の活性化関数は、確率化するためのものの処理と理解した。
- ・中間層と出力層は異なる活性化関数を利用するため、その組み合わせによって結果が異なると理解した。
- ・二乗誤差を用いる理由は、誤差をそのまま足し算すると 0 となってしまうから。また、二乗誤差に 1/2 がついているのは、微分時の計算が楽になるためのものということから、値そのものの以外にも計算の容易さが選定基準になることを知った。

## 4 勾配降下法

### 4.1 要点のまとめ

- ・勾配降下法を利用してパラメータを最適化  
誤差を最小化するパラメータを発見  
$$w^{(t+1)} = w^{(t)} - \epsilon \nabla E$$
- ・学習率の値によって学習の効率が大きく異なる。
- ・学習率が大きすぎると最小値にたどり着かず発散するリスクがある
- ・学習率の値によって学習の効率が大きく異なる。
- ・確率的勾配降下法のメリットは、計算コストの軽減、局所極小解に収束するリスクの削減、オンライン学習ができること。
- ・ミニバッチ勾配降下法のメリットは、確率的勾配降下法のメリットに加え、計算資源を有効利用できること。



## 4.2 演習結果

### ▼ 確率勾配降下法

```
# 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 = [{} 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)

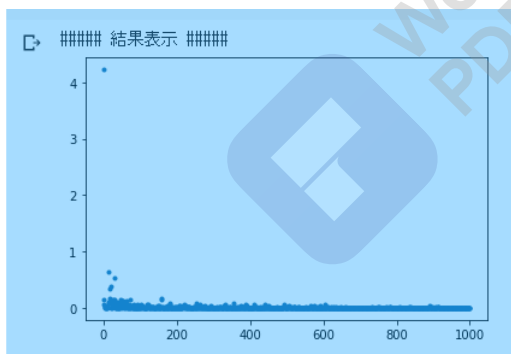
# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('w1', 'w2', 'b1', 'b2'):
        network[key] -= learning_rate * grad[key]

# 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()

```



### 4.3 考察

- ・適切な学習率を設定するためには、経験が必要だと感じた。
- ・オンライン学習とは、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、学習を進めていく。バッチ学習は一度にすべての学習データを使ってパラメータ更新を行う。
- ・ミニバッチはオンライン学習とバッチ学習の中間のようなもの。計算処理が大量なため、効率よく計算資源を利用できるように考えることが必要。

## 5 誤差逆伝播法

### 5.1 要点のまとめ

- ・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
- ・誤差関数には、二乗誤差関数が使用される。

### 5.2 演習結果

```
import numpy as np
from data.vnist import load_vnist
import pickle
from common import functions
import matplotlib.pyplot as plt

# mnistをロード
(x_train, y_train), (x_test, y_test) = load_vnist(normalize=True, one_hot_label=True)
train_size = len(x_train)

print("データ読み込み完了")

# 重み初期値補正係数
weight_init = 0.01 # 変更してみよう
# 入力層サイズ
input_layer_size = 784 # 変更してみよう
# 中間層サイズ
hidden_layer_size = 40 # 変更してみよう
# 出力層サイズ
output_layer_size = 10 # 変更してみよう
# 繰り返し数
iters_num = 1000 # 変更してみよう
# ミニバッチサイズ
batch_size = 100 # 変更してみよう
# 学習率
learning_rate = 0.1 # 変更してみよう
# 描画頻度
plot_interval=10

# 初期値設定
def init_network():
    network = {}
    network['W1'] = weight_init * np.random.randn(input_layer_size, hidden_layer_size)
    network['W2'] = weight_init * np.random.randn(hidden_layer_size, output_layer_size)
    # 試してみよう Xavierの初期値
    network['W1'] = np.random.randn(input_layer_size, hidden_layer_size) / np.sqrt(input_layer_size)
    network['W2'] = np.random.randn(hidden_layer_size, output_layer_size) / np.sqrt(hidden_layer_size)
    # 試してみよう Heの初期値
    network['W1'] = np.random.randn(input_layer_size, hidden_layer_size) / np.sqrt(input_layer_size) * np.sqrt(2)
    network['W2'] = np.random.randn(hidden_layer_size, output_layer_size) / np.sqrt(hidden_layer_size) * np.sqrt(2)
```

```

network['b1'] = np.zeros(hidden_layer_size)
network['b2'] = np.zeros(output_layer_size)

return network

# 順伝播
def forward(network, x):
    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)

    return z1, y

# 誤差逆伝播
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_softmax_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)
    # 1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)

    return grad

# パラメータの初期化
network = init_network()

accuracies_train = []
accuracies_test = []

```

```

# 正答率
def accuracy(x, d):
    z1, y = forward(network, x)
    y = np.argmax(y, axis=1)
    if d.ndim != 1 : d = np.argmax(d, axis=1)
    accuracy = np.sum(y == d) / float(x.shape[0])
    return accuracy

for i in range(1000):
    # ランダムにバッチを取得
    batch_mask = np.random.choice(train_size, batch_size)
    # ミニバッチに対応する教師訓練画像データを取得
    x_batch = x_train[batch_mask]
    # ミニバッチに対応する訓練正解ラベルデータを取得する
    d_batch = d_train[batch_mask]

    z1, y = forward(network, x_batch)
    grad = backward(x_batch, d_batch, z1, y)

    if (i%1000) == 0:
        accr_test = accuracy(x_test, d_test)
        accuracies_test.append(accr_test)

        accr_train = accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

        print('Generation: ' + str(i+1) + ', 正答率(トレーニング) = ' + str(accr_train))
        print('          : ' + str(i+1) + ', 正答率(テスト) = ' + str(accr_test))

    # パラメータに勾配の適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key] += learning_rate * grad[key]

lists = range(0, 1000, 1000)
plt.plot(lists, accuracies_train, label='training set')
plt.plot(lists, accuracies_test, label='test set')
plt.legend(loc='lower right')
plt.title("正答率")
# グラフの表示
plt.show()

```



### 5.3 考察

- ・微分を連続して掛け算することで、余分な計算をしなくてよい。
- ・入力層に近づくにしたがって、掛け算の項数が増える。
- ・偏微分の性質をうまく利用していると認識した。

## 深層学習 day2 レポート

### 1 勾配消失問題

#### 1.1 要点のまとめ

・誤差逆伝播法が下位層に進んでいくに連れて、勾配が緩やかになる。そのため、勾配降下法において下位層のパラメータはほとんど変わらず、訓練は最適値に収束しなくなることもある。

・最も使われている活性化関数は ReLU 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上げている。

・Xavier の初期値を設定する際の活性化関数は下記のようなものがある。

ReLU 関数

シグモイド（ロジスティック）関数

双曲線正接関数

・He の初期値を設定する際の活性化関数は ReLU 関数を使用される。

#### 1.2 演習結果

sigmoid - gauss

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01)

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

    if (i + 1) % plot_interval == 0:
        accr_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accr_test)
        accr_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accr_train)

    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accr_train))
    print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accr_test))
```

```

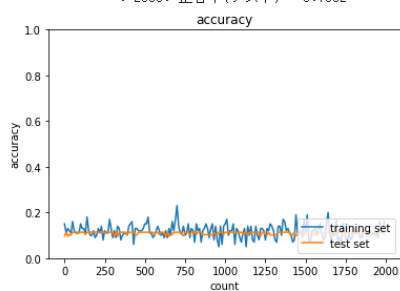
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

: 1900. 正答率(テスト) = 0.1135
Generation: 1960. 正答率(トレーニング) = 0.09
: 1960. 正答率(テスト) = 0.1135
Generation: 1970. 正答率(トレーニング) = 0.18
: 1970. 正答率(テスト) = 0.1135
Generation: 1980. 正答率(トレーニング) = 0.12
: 1980. 正答率(テスト) = 0.1135
Generation: 1990. 正答率(トレーニング) = 0.12
: 1990. 正答率(テスト) = 0.1135
Generation: 2000. 正答率(トレーニング) = 0.18
: 2000. 正答率(テスト) = 0.1032

```



#### ▼ ReLU - gauss

```

# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='relu', weight_init_std=0.01)

iters_num = 2000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.1

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)

    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        network.params[key] -= learning_rate * grad[key]

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

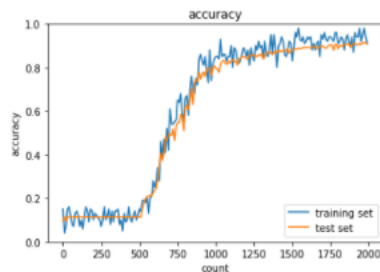
    if (i + 1) % plot_interval == 0:
        accor_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accor_test)
        accor_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accor_train)

        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accor_train))
        print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accor_test))

lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```
Generation: 1910. 正答率(トレーニング) = 0.87
                  : 1910. 正答率(テスト) = 0.9086
Generation: 1920. 正答率(トレーニング) = 0.96
                  : 1920. 正答率(テスト) = 0.911
Generation: 1930. 正答率(トレーニング) = 0.94
                  : 1930. 正答率(テスト) = 0.9023
Generation: 1940. 正答率(トレーニング) = 0.94
                  : 1940. 正答率(テスト) = 0.9053
Generation: 1950. 正答率(トレーニング) = 0.98
                  : 1950. 正答率(テスト) = 0.91
Generation: 1960. 正答率(トレーニング) = 0.92
                  : 1960. 正答率(テスト) = 0.9043
Generation: 1970. 正答率(トレーニング) = 0.95
                  : 1970. 正答率(テスト) = 0.9145
Generation: 1980. 正答率(トレーニング) = 0.98
                  : 1980. 正答率(テスト) = 0.915
Generation: 1990. 正答率(トレーニング) = 0.94
                  : 1990. 正答率(テスト) = 0.9119
Generation: 2000. 正答率(トレーニング) = 0.91
                  : 2000. 正答率(テスト) = 0.9073
```



### 1.3 考察

- ・連鎖律の原理では、微分を偏微分の掛け算で示すことができる。
- ・シグモイド関数は緩やかに変化するため信号の強弱を伝えるには有用であったが、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。
- ・シグモイド関数はニューラルネットワークの導入に大きく貢献したが、勾配消失問題があり、他の関数が使われることが多くなった。



## 2 学習率最適化手法

### 2.1 要点のまとめ

- ・ 学習率は大きい場合発散しやすく。小さい場合は収束しづらくなる。
- ・ 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ことや、パラメータごとに学習率を可変させることが初期の学習率設定方法の指針
- ・ 学習率最適化手法として下記がある。

- ① モメンタム
- ② AdaGrad
- ③ RMSProp
- ④ Adam

### 2.2 演習結果

#### ▼ Momentum

```
# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True, one_hot_label=True)

print("データ読み込み完了")

# batch_normalizationの設定 =====
# use_batchnorm = True
use_batchnorm = False
# =====

network = MultiLayerNet(input_size=784, hidden_size_list=[40, 20], output_size=10, activation='sigmoid', weight_init_std=0.01,
                        use_batchnorm=use_batchnorm)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate = 0.3
# 慣性
momentum = 0.9

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    # 勾配
    grad = network.gradient(x_batch, d_batch)
    if i == 0:
        v = {}
    for key in ('W1', 'W2', 'W3', 'b1', 'b2', 'b3'):
        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]

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

    if (i + 1) % plot_interval == 0:
        accor_test = network.accuracy(x_test, d_test)
        accuracies_test.append(accor_test)
        accor_train = network.accuracy(x_batch, d_batch)
        accuracies_train.append(accor_train)

    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(accor_train))
    print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(accor_test))
```

```

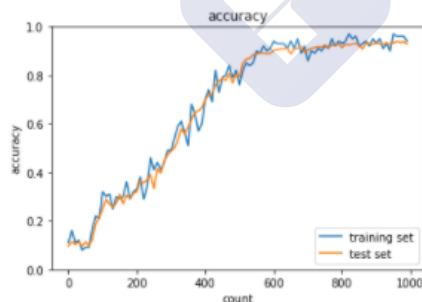
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

: 800. 正答率(テスト) = 0.9281
Generation: 810. 正答率(トレーニング) = 0.93
: 810. 正答率(テスト) = 0.9135
Generation: 820. 正答率(トレーニング) = 0.94
: 820. 正答率(テスト) = 0.9285
Generation: 830. 正答率(トレーニング) = 0.97
: 830. 正答率(テスト) = 0.9245
Generation: 840. 正答率(トレーニング) = 0.95
: 840. 正答率(テスト) = 0.9289
Generation: 850. 正答率(トレーニング) = 0.96
: 850. 正答率(テスト) = 0.9306
Generation: 860. 正答率(トレーニング) = 0.92
: 860. 正答率(テスト) = 0.9231
Generation: 870. 正答率(トレーニング) = 0.93
: 870. 正答率(テスト) = 0.9073
Generation: 880. 正答率(トレーニング) = 0.94
: 880. 正答率(テスト) = 0.9313
Generation: 890. 正答率(トレーニング) = 0.92
: 890. 正答率(テスト) = 0.9295
Generation: 900. 正答率(トレーニング) = 0.95
: 900. 正答率(テスト) = 0.927
Generation: 910. 正答率(トレーニング) = 0.93
: 910. 正答率(テスト) = 0.9333
Generation: 920. 正答率(トレーニング) = 0.95
: 920. 正答率(テスト) = 0.929
Generation: 930. 正答率(トレーニング) = 0.91
: 930. 正答率(テスト) = 0.9279
Generation: 940. 正答率(トレーニング) = 0.93
: 940. 正答率(テスト) = 0.9333
Generation: 950. 正答率(トレーニング) = 0.9
: 950. 正答率(テスト) = 0.926
Generation: 960. 正答率(トレーニング) = 0.97
: 960. 正答率(テスト) = 0.9325
Generation: 970. 正答率(トレーニング) = 0.96
: 970. 正答率(テスト) = 0.939
Generation: 980. 正答率(トレーニング) = 0.96
: 980. 正答率(テスト) = 0.9354
Generation: 990. 正答率(トレーニング) = 0.96
: 990. 正答率(テスト) = 0.9378
Generation: 1000. 正答率(トレーニング) = 0.94
: 1000. 正答率(テスト) = 0.9285

```



## 2.3 考察

- ・モメンタムは、局所最適解に陥らずに大域最適解になりやすい。
- ・AdaGrad は勾配の緩やかな斜面に対して最適値に近づけることができるが、鞍点問題を引き起こすことがある。
- ・RMSProp は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
- ・Adam はモメンタムおよび RMSProp のメリットを孕んだアルゴリズム

### 3 過学習

#### 3.1 要点のまとめ

- ・ネットワークの自由度(層数、ノード数、パラメータの値 etc...)を制約し過学習を抑制することを正則化という。
- ・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。過学習をおさえるため、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。
- ・誤差関数に p ノルムを追加し L1、L2 正則化を行う。P=1 の場合 L1 正則化、P=2 の場合 L2 正則化と呼ぶ。
- ・ノード数の多さが過学習の課題であり、ランダムにノードを削除して学習させることをドロップアウトという。

#### 3.2 演習結果

▼ weight decay

L2

```
from common import optimizer

(x_train, d_train), (x_test, d_test) = load_mnist(normalize=True)

print("データ読み込み完了")

# 過学習を再現するために、学習データを削減
x_train = x_train[:300]
d_train = d_train[:300]

network = MultiLayerNet(input_size=784, hidden_size_list=[100, 100, 100, 100, 100, 100], output_size=10)

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100
learning_rate=0.01

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10
hidden_layer_num = network.hidden_layer_num

# 正則化強度設定
weight_decay_lambda = 0.1

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    weight_decay = 0

    for idx in range(1, hidden_layer_num+1):
        grad['W' + str(idx)] = network.layers['Affine' + str(idx)].dW + weight_decay_lambda * network.params['W' + str(idx)]
        grad['b' + str(idx)] = network.layers['Affine' + str(idx)].db
        network.params['W' + str(idx)] -= learning_rate * grad['W' + str(idx)]
        network.params['b' + str(idx)] -= learning_rate * grad['b' + str(idx)]
        weight_decay += 0.5 * weight_decay_lambda * np.sqrt(np.sum(network.params['W' + str(idx)] ** 2))

    loss = network.loss(x_batch, d_batch) + weight_decay
    train_loss_list.append(loss)

    if (i+1) % plot_interval == 0:
        acor_train = network.accuracy(x_train, d_train)
        acor_test = network.accuracy(x_test, d_test)
        accuracies_train.append(acor_train)
        accuracies_test.append(acor_test)

    print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acor_train))
    print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(acor_test))
```

```

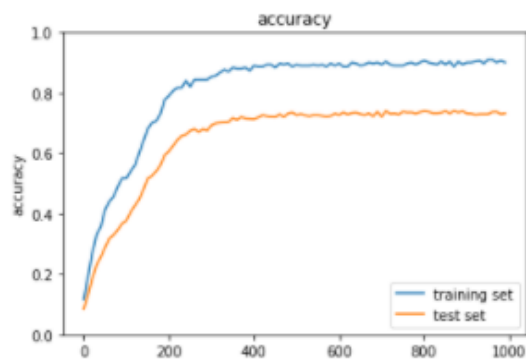
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

: 800. 正答率(テスト) = 0.7356
Generation: 810. 正答率(トレーニング) = 0.9066666666666666
: 810. 正答率(テスト) = 0.7393
Generation: 820. 正答率(トレーニング) = 0.9
: 820. 正答率(テスト) = 0.737
Generation: 830. 正答率(トレーニング) = 0.8966666666666666
: 830. 正答率(テスト) = 0.7324
Generation: 840. 正答率(トレーニング) = 0.8933333333333333
: 840. 正答率(テスト) = 0.7322
Generation: 850. 正答率(トレーニング) = 0.9033333333333333
: 850. 正答率(テスト) = 0.7337
Generation: 860. 正答率(トレーニング) = 0.8933333333333333
: 860. 正答率(テスト) = 0.7393
Generation: 870. 正答率(トレーニング) = 0.9033333333333333
: 870. 正答率(テスト) = 0.7312
Generation: 880. 正答率(トレーニング) = 0.8866666666666667
: 880. 正答率(テスト) = 0.74
Generation: 890. 正答率(トレーニング) = 0.9033333333333333
: 890. 正答率(テスト) = 0.7346
Generation: 900. 正答率(トレーニング) = 0.8933333333333333
: 900. 正答率(テスト) = 0.7389
Generation: 910. 正答率(トレーニング) = 0.9
: 910. 正答率(テスト) = 0.7314
Generation: 920. 正答率(トレーニング) = 0.9
: 920. 正答率(テスト) = 0.7308
Generation: 930. 正答率(トレーニング) = 0.9033333333333333
: 930. 正答率(テスト) = 0.7272
Generation: 940. 正答率(トレーニング) = 0.9066666666666666
: 940. 正答率(テスト) = 0.7288
Generation: 950. 正答率(トレーニング) = 0.8966666666666666
: 950. 正答率(テスト) = 0.7289
Generation: 960. 正答率(トレーニング) = 0.91
: 960. 正答率(テスト) = 0.729
Generation: 970. 正答率(トレーニング) = 0.91
: 970. 正答率(テスト) = 0.7362
Generation: 980. 正答率(トレーニング) = 0.9
: 980. 正答率(テスト) = 0.7369
Generation: 990. 正答率(トレーニング) = 0.9066666666666666
: 990. 正答率(テスト) = 0.7292
Generation: 1000. 正答率(トレーニング) = 0.9
: 1000. 正答率(テスト) = 0.7317

```



### 3.3 考察

- ・過学習の状態、改善傾向を確認する方法として学習曲線を確認する方法がある。
- ・過学習を抑える手法は多数あるが、どの手法が良いかについては、データとの相性もあると感じた。
- ・過学習の抑制についてすべてを自動化するのは難しく、モデル作成する人間の判断が必要なものもまだまだあると感じた。



## 4 畳み込みニューラルネットワークの概念

### 4.1 要点のまとめ

- ・畳み込み層では画像の場合、縦、横、チャンネルの3次元の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。
- ・パディング、ストライドは畳み込み演算の処理として特徴的な処理である。
- ・CNNの構造は、入力層、畳み込み層、プーリング層、全結合層、出力層から構成される。

### 4.2 演習結果

#### ▼ simple convolution network class

```
[8] class SimpleConvNet:
    # conv - relu - pool - affine - relu - affine - softmax
    def __init__(self, input_dim=(1, 28, 28), conv_param={'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        filter_num = conv_param['filter_num']
        filter_size = conv_param['filter_size']
        filter_pad = conv_param['pad']
        filter_stride = conv_param['stride']
        input_size = input_dim[1]
        conv_output_size = (input_size - filter_size + 2 * filter_pad) / filter_stride + 1
        pool_output_size = int(filter_num * (conv_output_size / 2) * (conv_output_size / 2))

        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(filter_num, input_dim[0], filter_size, filter_size)
        self.params['b1'] = np.zeros(filter_num)
        self.params['W2'] = weight_init_std * np.random.randn(pool_output_size, hidden_size)
        self.params['b2'] = np.zeros(hidden_size)
        self.params['W3'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b3'] = np.zeros(output_size)

        # レイヤの生成
        self.layers = OrderedDict()
        self.layers['Conv1'] = layers.Convolution(self.params['W1'], self.params['b1'], conv_param['stride'], conv_param['pad'], conv_param['filter_size'])
        self.layers['Relu1'] = layers.ReLU()
        self.layers['Pool1'] = layers.Pooling(pool_h=2, pool_w=2, stride=2)
        self.layers['Affine1'] = layers.Affine(self.params['W2'], self.params['b2'])
        self.layers['Relu2'] = layers.ReLU()
        self.layers['Affine2'] = layers.Affine(self.params['W3'], self.params['b3'])

        self.last_layer = layers.SoftmaxWithLoss()

    def predict(self, x):
        for key in self.layers.keys():
            x = self.layers[key].forward(x)
        return x

    def loss(self, x, d):
        y = self.predict(x)
        return self.last_layer.forward(y, d)

    def accuracy(self, x, d, batch_size=100):
        if d.ndim != 1 : d = np.argmax(d, axis=1)
        acc = 0.0

        for i in range(int(x.shape[0] / batch_size)):
            tx = x[i*batch_size:(i+1)*batch_size]
            td = d[i*batch_size:(i+1)*batch_size]
            y = self.predict(tx)
            y = np.argmax(y, axis=1)
            acc += np.sum(y == td)

        return acc / x.shape[0]
```

```
dout = self.last_layer.backward(dout)
layers = list(self.layers.values())

layers.reverse()
for layer in layers:
    dout = layer.backward(dout)

# 設定
grad = {}
grad['W1'], grad['b1'] = self.layers['Conv1'].dW, self.layers['Conv1'].db
grad['W2'], grad['b2'] = self.layers['Affine1'].dW, self.layers['Affine1'].db
grad['W3'], grad['b3'] = self.layers['Affine2'].dW, self.layers['Affine2'].db

return grad
```

```
from common import optimizer

# データの読み込み
(x_train, d_train), (x_test, d_test) = load_mnist(flatten=False)

print("データ読み込み完了")

# 処理に時間のかかる場合はデータを削減
x_train, d_train = x_train[:5000], d_train[:5000]
x_test, d_test = x_test[:1000], d_test[:1000]

network = SimpleConvNet(input_dim=(1,28,28), conv_param = {'filter_num': 30, 'filter_size': 5, 'pad': 0, 'std': 0.01, 'hidden_size': 100, 'output_size': 10, 'weight_init_std': 0.01})

optimizer = optimizer.Adam()

iters_num = 1000
train_size = x_train.shape[0]
batch_size = 100

train_loss_list = []
accuracies_train = []
accuracies_test = []

plot_interval=10

for i in range(iters_num):
    batch_mask = np.random.choice(train_size, batch_size)
    x_batch = x_train[batch_mask]
    d_batch = d_train[batch_mask]

    grad = network.gradient(x_batch, d_batch)
    optimizer.update(network.params, grad)

    loss = network.loss(x_batch, d_batch)
    train_loss_list.append(loss)

    if (i+1) % plot_interval == 0:
        acoor_train = network.accuracy(x_train, d_train)
        acoor_test = network.accuracy(x_test, d_test)
        accuracies_train.append(acoor_train)
        accuracies_test.append(acoor_test)

        print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acoor_train))
        print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(acoor_test))
```

```

print('Generation: ' + str(i+1) + '. 正答率(トレーニング) = ' + str(acor_train))
print('          : ' + str(i+1) + '. 正答率(テスト) = ' + str(acor_test))

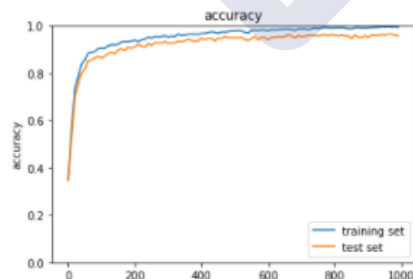
lists = range(0, iters_num, plot_interval)
plt.plot(lists, accuracies_train, label="training set")
plt.plot(lists, accuracies_test, label="test set")
plt.legend(loc="lower right")
plt.title("accuracy")
plt.xlabel("count")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
# グラフの表示
plt.show()

```

```

: 800. 正答率(テスト) = 0.963
Generation: 810. 正答率(トレーニング) = 0.991
: 810. 正答率(テスト) = 0.961
Generation: 820. 正答率(トレーニング) = 0.9918
: 820. 正答率(テスト) = 0.956
Generation: 830. 正答率(トレーニング) = 0.9922
: 830. 正答率(テスト) = 0.961
Generation: 840. 正答率(トレーニング) = 0.9908
: 840. 正答率(テスト) = 0.957
Generation: 850. 正答率(トレーニング) = 0.9872
: 850. 正答率(テスト) = 0.957
Generation: 860. 正答率(トレーニング) = 0.9868
: 860. 正答率(テスト) = 0.945
Generation: 870. 正答率(トレーニング) = 0.9918
: 870. 正答率(テスト) = 0.961
Generation: 880. 正答率(トレーニング) = 0.9918
: 880. 正答率(テスト) = 0.952
Generation: 890. 正答率(トレーニング) = 0.9906
: 890. 正答率(テスト) = 0.951
Generation: 900. 正答率(トレーニング) = 0.991
: 900. 正答率(テスト) = 0.958
Generation: 910. 正答率(トレーニング) = 0.9914
: 910. 正答率(テスト) = 0.951
Generation: 920. 正答率(トレーニング) = 0.9928
: 920. 正答率(テスト) = 0.959
Generation: 930. 正答率(トレーニング) = 0.9932
: 930. 正答率(テスト) = 0.96
Generation: 940. 正答率(トレーニング) = 0.994
: 940. 正答率(テスト) = 0.959
Generation: 950. 正答率(トレーニング) = 0.9956
: 950. 正答率(テスト) = 0.964
Generation: 960. 正答率(トレーニング) = 0.995
: 960. 正答率(テスト) = 0.96
Generation: 970. 正答率(トレーニング) = 0.996
: 970. 正答率(テスト) = 0.965
Generation: 980. 正答率(トレーニング) = 0.9948
: 980. 正答率(テスト) = 0.965
Generation: 990. 正答率(トレーニング) = 0.9956
: 990. 正答率(テスト) = 0.96
Generation: 1000. 正答率(トレーニング) = 0.9942
: 1000. 正答率(テスト) = 0.957

```



### 4.3 考察

- ・全結合層のデメリットは、RGB の各チャンネル間の関連性が、学習に反映されないことである。
- ・プーリング層の処理は畳み込み層の処理と比べ単純な計算が多く、対象領域の Max 値または、平均値を取得して出力する方法がある。
- ・全結合層のデメリットは、RGB の各チャンネル間の関連性が、学習に反映されないということである。



## 5 最新の CNN

### 5.1 要点のまとめ

- ・ AlexNet は ImageNet を解くためのモデル。
- ・ AlexNet は 5 層の畳み込み層及びプーリング層、それに続く 3 層の全結合層からなる。
- ・ 全結合層の前では  $13 \times 13 \times 256$  のデータになっている。
- ・ 全結合の処理として Fratten の処理は、そのまま縦に並べる処理。初期のニューラルネットワークでは、比較的よく使われる。

### 5.2 考察

- ・ GlobalMaxPooling や GlobalAveragePooling は Fratten に比べて非常に単純な処理ではあるが、なぜか精度がよくなる。
- ・ ニューラルネットワークの世界では、論理的な説明ができないこともあると再認識した。
- ・ 人の画像認識については、個人所法保護や人種問題等倫理的な問題もあり、慎重にすすめなければならない。

### 5.3 関連記事レポート

- ・ ILSVRC は大規模画像認識の競技会で AlexNet の他、ZFNet、GooLnet、RESNet、SENet 等の CNN モデルが優秀な成績を収めている。
- ・ ResNet が 1 つの大きなブレイクスルーであり、それ以降のモデルはほとんどが ResNet の改良と言える。
- ・ 組み込みデバイスや、スマートフォン等の、計算資源が潤沢ではない環境においては、高速に動作するモデルが必要であり、その一例が SqueezeNet である。