

深層学習 day2 レポート

復習

- 1: 入力層（入力ノード）に値を入力する
- 2: 重み、バイアス、活性化関数で計算する。（入力層→中間層→出力層）
- 3: 出力層から値が伝わる
- 4: 出力層から出た値と正解値から、誤差関数を使って誤差を求める
- 5: 誤差を小さくするため重みやバイアスを更新（学習）する。

Section1: 勾配消失問題

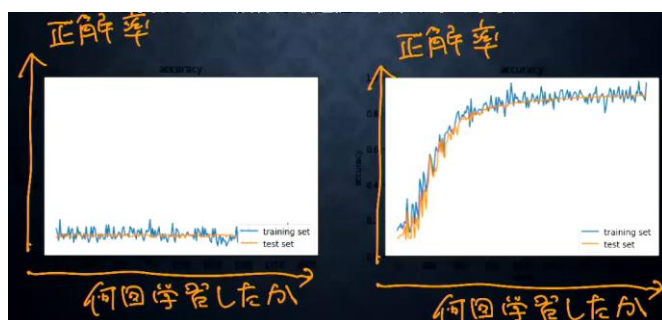
出力値と正解値から誤差関数を用いて誤差を計算。

その結果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。

勾配消失問題

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、訓練は最適値に収束しなくなる。（下図 左が勾配消失あり、右が通常の学習）



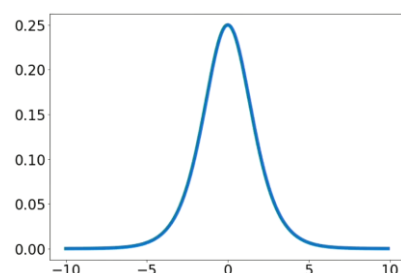
なぜ勾配消失が発生するか。

- ・ 微分結果が 1 以下となる
- ・ 中間層が増えた場合に微分が増えている。

上記の場合に連鎖律より 1 以下の掛け合わせが起こり、0 に近づくことになる。

微分結果が 1 以下になる理由に、シグモイド関数がある。

シグモイド関数の微分結果を下図に示す。



上記に示すように、最大で 0.25 となっておりシグモイド関数を用いた多層ニューラルネットワークでは、学習時に勾配消失が発生しやすい。

勾配消失の解決方

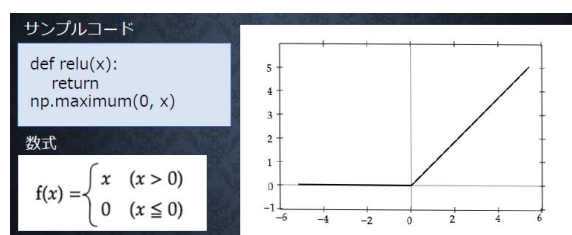
・活性化関数

● ReLU 関数

今最も使われている活性化関数

勾配消失問題の回避とスパース化に貢献することで良い成果をもたらしている。

(必要な重みだけのこり、不要な重みは取り除かれる)



・重みの初期値設定

● Xavier . . . シグモイド関数など S 字カーブの活性化関数に適用

ランダムに重みを設定する。(標準正規分布)

重みの要素を、前の層のノード数の平方根で除算した値

```
# 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)
```

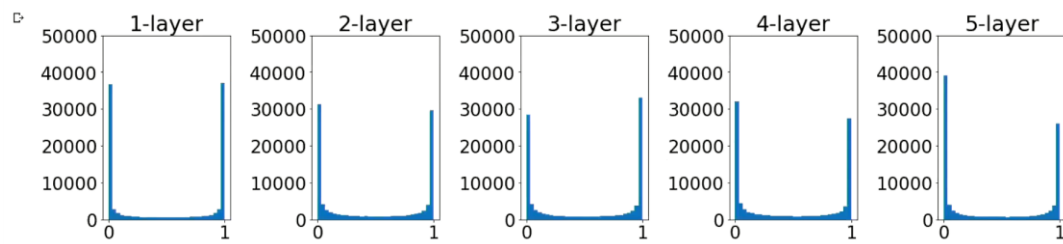
5 つの中間層を持つニューラルネットワークを考えてみる。

標準正規分布で重みを初期化したとき、各レイヤーの出力は 0 と 1 に偏る。

0 や 1 のときは微分値がほとんど 0 となる。(シグモイド関数微分図を参照)

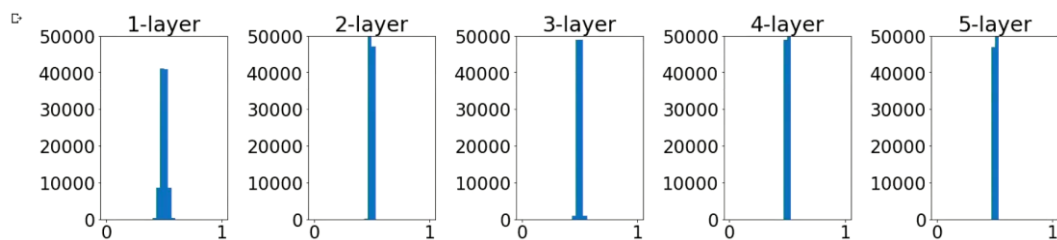
そのため誤差逆伝播によるパラメータ更新が行われなくなる。

```
[6] 1 show_activation(sigmoid, lambda n: np.random.randn(n, n) * 1)
```



次に、標準誤差を 0.01 にした正規分布で重みを初期化した場合。

```
[7] 1 show_activation(sigmoid, lambda n: np.random.randn(n, n) * 0.01)
```



この場合は出力値は 0.5 付近に集中するため、勾配消失は発生しづらくなる。

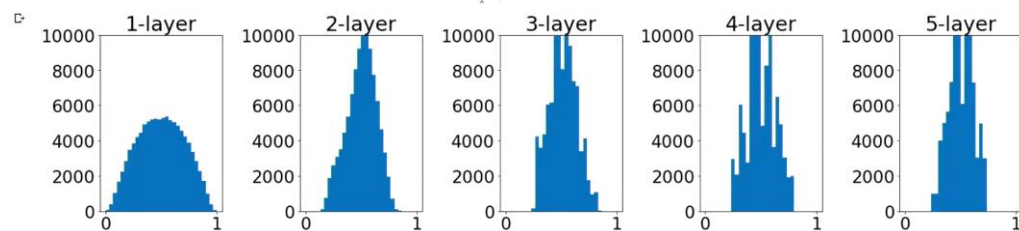
一方で折角活性化関数を通った値にもかかわらず、ほぼ 0 付近に集中し、表現力は損なわれている。

Xavier 初期化の場合

各レイヤーを通った後の値はどれもある程度バラツキを持ち、偏りはない。

そのため活性化関数の表現力を保ちつつ、勾配消失への対策となっている。

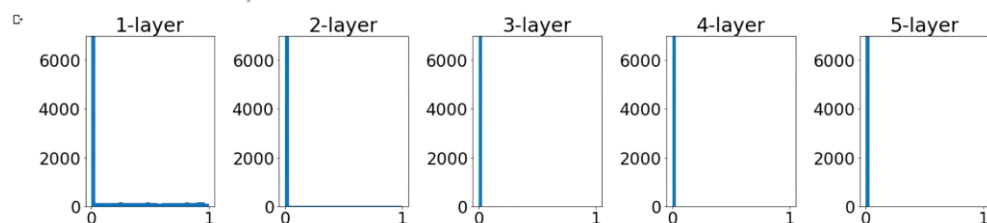
```
[8] 1 show_activation(sigmoid, lambda n: np.random.randn(n, n) * np.sqrt(1.0 / n), (0, 10000))
```



● He …… ReLU 関数など S 字カーブの活性化関数に適用

標準正規分布に基づいた重みを用いて ReLU 関数を通すと表現力がない。(0 か 1 に偏る)

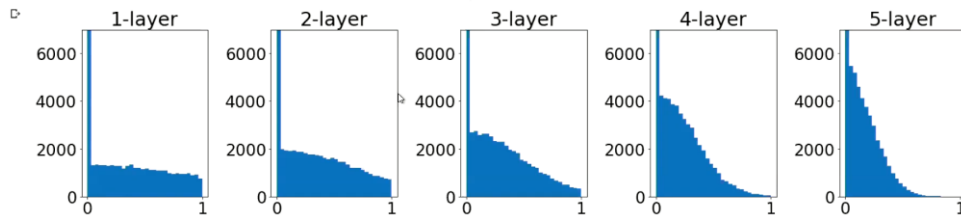
```
[9] 1 show_activation(reLU, lambda n: np.random.randn(n, n) * 1, (0, 7000))
```



He 初期化を行った場合

ある程度の表現力を保った状態となる。

```
[11] 1 show_activation(relu, lambda ni: np.random.randn(n, n) * np.sqrt(1.0 / n), (0, 7000))
```



• 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

バッチ正規化の使い所とは？

活性化関数に値を渡す前後に、バッチ正規化の処理を孕んだ層を加える

ミニバッチのサイズについては、機材による

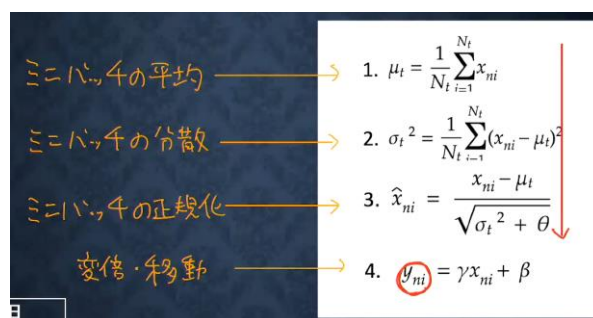
CPU・・・汎用

GPU・・・ゲーム用/AI（1～64枚）

TPU・・・AI専用（1～256枚）

1エポックではミニバッチサイズで学習を行うことになる。

ミニバッチ正規化の数学的記述



μ_t : ミニバッチ全体の平均
 σ_t^2 : ミニバッチ全体の標準偏差
 N_t : ミニバッチのインデックス
 \hat{x}_{ni} : 0に値を近づける計算(を中心とするセントラリング)と正規化を施した値
 γ : スケーリングパラメータ
 β : シフトパラメータ
 y_{ni} : ミニバッチのインデックス値とスケーリングの積にシフトを加算した値(バッチ正規化オペレーションの出力)

• 確認テスト

[連鎖律の原理を使い、dz/dx を求めよ。]

$$z = t^2$$

$$t = x + y$$

解答： $\frac{dz}{dx} = \frac{dz}{dt} \frac{dt}{dx} = 2t \cdot 1 = 2(x + y)$

[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 (解答)
- (3) 0.35
- (4) 0.45

[重みの初期値に 0 を設定すると、どのような問題が発生するか。簡潔に説明せよ。]

解答：ニューラルネットワークの表現力が無い状態と考える。

(補足) 全て均一に学習するとの講師談。

[一般的に考えられるバッチ正規化の効果を 2 点挙げよ。]

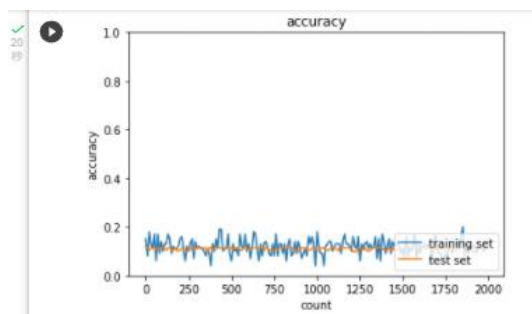
解答：中間層の重みの更新がうまくいき、モデルの収束が早くなる。

過学習の抑制などが考えられる。

・実装演習

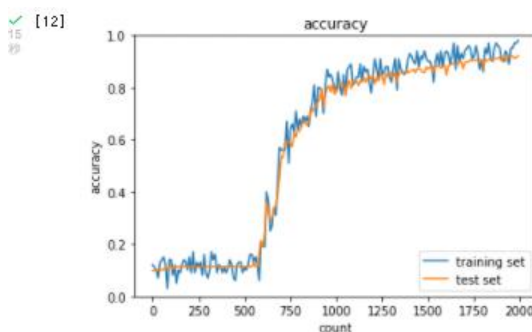
2_2_2_vanishing_gradient_modified.ipynb を実行

vanishing gradient modified . . . 失敗する学習方法



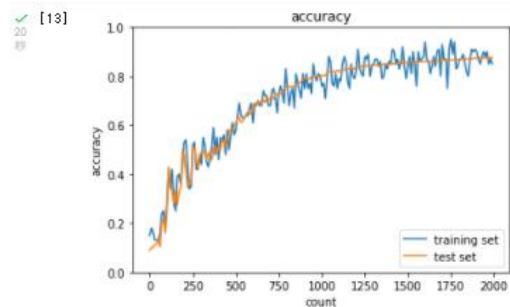
学習が進まないことが現れている。

ReLU - gauss . . . ReLU は 0.5 近傍は傾き 1 で特に問題ないが。



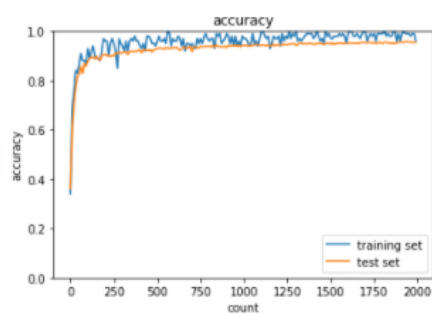
初期の学習が停滞している。しかし、学習完

sigmoid – Xavier . . . 本命の学習方法



きちんと学習が進む。

ReLU – He . . . こちらも本命の学習方法



gauss に対しスムーズに学習できている。

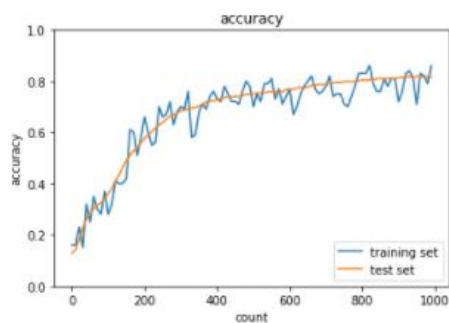
2_3_batch_normalization.ipynb

sys.path の設定

私の環境 (Path) では、commom も追加する必要があったので、ご参考に。

```
[9] 1 import sys
2 # sys.path.append('/content/drive/My Drive/DNN_code')
3 # sys.path.append('/content/drive/My Drive/DNN_code/lesson_2')
4 sys.path.append('/content/drive/My Drive/StudyAI/Step3')
5 sys.path.append('/content/drive/My Drive/StudyAI/Step3/common')
6 sys.path.append('/content/drive/My Drive/StudyAI/Step3/lesson_2')
```

batch normalization



うまく学習ができている。

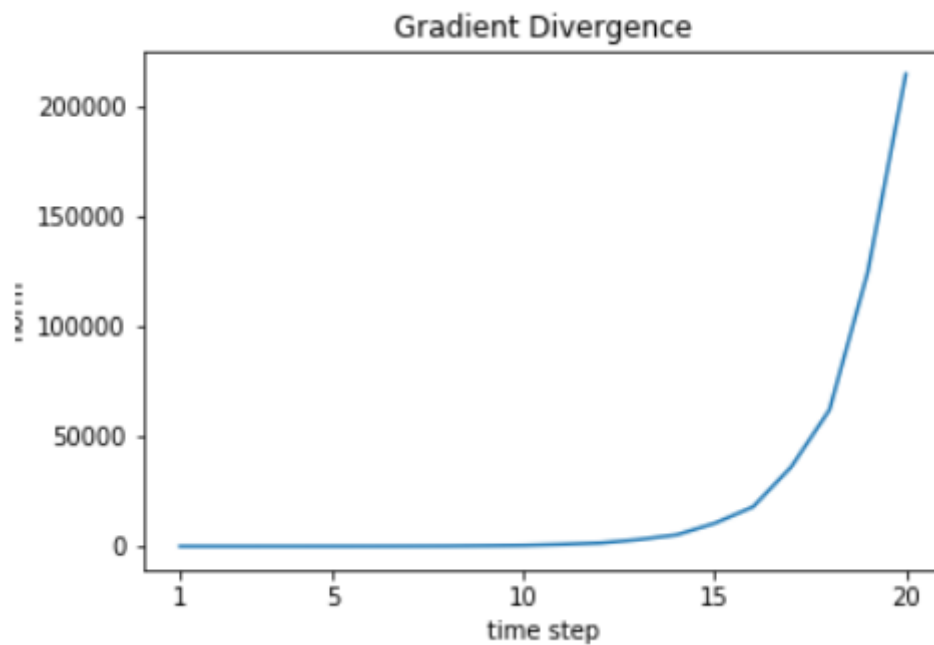
batch 構成で学習の多少の影響は出ている？

・その他

講義では勾配爆発（勾配発散）問題について触れていないため補足として調べた結果を載せる。

勾配爆発問題は、勾配が大きくなりすぎて学習が発散する問題。
ニューラルネットワークでは行列の積によりモデル表現しており、
重みが過剰に大きくなると、その出力による依存が大きすぎモデル表現が阻害される。
シンプルな RNN でも勾配爆発は発生するため注意が必要。

RNN では、決まった時間サイズの行列積と活性化関数の演算を繰り返す。
そのため簡単に勾配爆発が発生するメカニズムとなる。



※ [マサムネの部屋](#)より参照： RNN による勾配グラフ

学習により勾配がどんどん大きくなっている。

Section2：学習率最適化手法

勾配降下法における学習率（ ϵ ）

大きい：最適値にいつまでもたどり着かず発散してしまう。

小さい：発散することはないが、収束するまでに時間がかかる。

大域局所最適値に収束しづらくなる

従って、最適な学習率の設定が必要不可欠となる。

学習率最適化手法を利用して学習率を最適化を行う。

● モメンタム

- ・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も低い位置(最適値)にいくまでの時間が早い。

| 【モメンタム】 | 【勾配降下法】 |
|---|---|
| $V_t = \mu V_{t-1} - \epsilon \nabla E$ | $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$ |
| $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + V_t$ | |
| 慣性： μ | |
| モメンタム | 勾配降下法 |
| 誤差をパラメータで微分したものと学習率の積を減算した後、現在の重みに前回の重みを減算した値と慣性の積を加算する | 誤差をパラメータで微分したものと学習率の積を減算する |

確率的勾配降下法での振動を抑える（移動平均のような動き）

動いたら動きつ続けたい特徴（なかなか安定しない）

● AdaGrad

- ・勾配の緩やかな斜面に対して、最適値に近づける

| 【AdaGrad】 | 【勾配降下法】 |
|---|---|
| $h_0 = \theta$ | $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$ |
| $h_t = h_{t-1} + (\nabla E)^2$ | |
| $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$ | |
| AdaGrad | 勾配降下法 |
| 誤差をパラメータで微分したもの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ものと学習率の積を減算する |

学習率が徐々に小さくなるので、鞍点問題を引き起こす事があった。

- RMSProp

- ・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

| 【RMSProp】 | 【勾配降下法】 |
|--|---|
| $h_t = \alpha h_{t-1} + (1 - \alpha) (\nabla E)^2$ $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \frac{1}{\sqrt{h_t + \theta}} \nabla E$ | $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E$ |
| RMSProp | 勾配降下法 |
| 誤差をパラメータで微分したもの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ものと学習率の積を減算する |

AdaGrad を改良したモデルため似たような動きとなる。

- Adam

- ・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
- 上記をそれぞれ孕んだ最適化アルゴリズムである。

- ・確認テスト

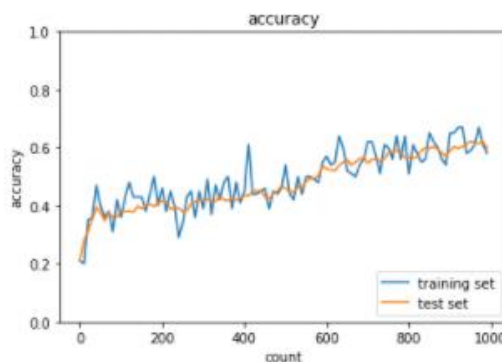
[モメンタム・AdaGrad・RMSProp の特徴をそれぞれ簡潔に説明せよ。]

- ・実装演習

2_4_optimizer.ipynb

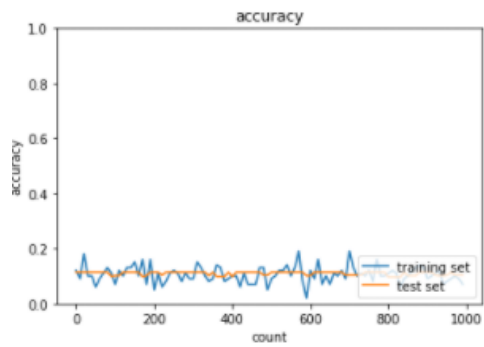
[try] バッチ正規化をしてみよう

- SGD



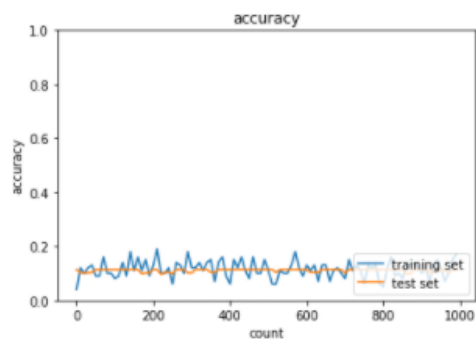
学習は進むが遅い

- Momentum



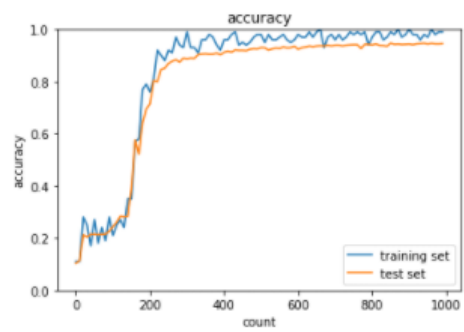
学習が進んでいない。

- AdaGrad



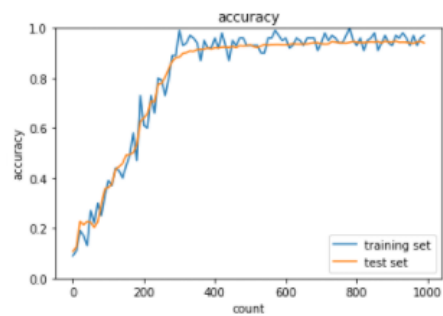
モメンタムベースのため同じく学習しない。

- RMSprop



学習初期は停滞するも、うまく学習できる。

- Adam



うまく学習できている。

- その他の最適化手法について（参照：[深層学習の最適化アルゴリズム](#)）

- ネステロフの加速法（NAG）（1983）

Momentum SGD の修正版であり，さらに収束への加速を増したものである．

$$\begin{aligned}\mathbf{g}^{(t)} &= \nabla E(\mathbf{w}^{(t)} + \mu \mathbf{w}^{(t-1)}) \\ \Delta \mathbf{w}^{(t)} &= \mu \Delta \mathbf{w}^{(t-1)} - (1 - \mu) \eta \mathbf{g}^{(t)} \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \Delta \mathbf{w}^{(t)}\end{aligned}$$

- RMSpropGraves (2014)

Graves が提案した RMSprop の改良版アルゴリズムである。

主に手書き文字認識の分野で用いられている。

$$\begin{aligned}\mathbf{g}^{(t)} &= \nabla E(\mathbf{w}^{(t)}) \\ \mathbf{m}_t &= \rho \mathbf{m}_{t-1} + (1 - \rho) \mathbf{g}^{(t)} \\ \mathbf{v}_t &= \rho \mathbf{v}_{t-1} + (1 - \rho) (\mathbf{g}^{(t)})^2 \\ \Delta \mathbf{w}^{(t)} &= -\frac{\eta}{\sqrt{\mathbf{v}_t - \mathbf{m}_t^2 + \varepsilon}} \mathbf{g}^{(t)} \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \Delta \mathbf{w}^{(t)}\end{aligned}$$

- SMORMS3 (2015)

RMSprop の問題点を曲率と LeCun 法によって克服したものである。

$$\begin{aligned}\mathbf{g}^{(t)} &= \nabla E(\mathbf{w}^{(t)}) \\ \mathbf{s}_t &= 1 + (1 - \zeta_{t-1} \mathbf{s}_{t-1}) \\ \rho_t &= \frac{1}{\mathbf{s}_t + 1} \\ \mathbf{m}_t &= \rho_t \mathbf{m}_{t-1} + (1 - \rho_t) \mathbf{g}^{(t)} \\ \mathbf{v}_t &= \rho_t \mathbf{v}_{t-1} + (1 - \rho_t) (\mathbf{g}^{(t)})^2 \\ \zeta_t &= \frac{\mathbf{m}_t^2}{\mathbf{v}_t + \varepsilon} \\ \Delta \mathbf{w}^{(t)} &= -\frac{\min\{\eta, \zeta_t\}}{\sqrt{\mathbf{v}_t + \varepsilon}} \mathbf{g}^{(t)} \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \Delta \mathbf{w}^{(t)}\end{aligned}$$

- AdaMax (2015)

無限次元ノルムに対応させたものが AdaMax である。

$$\begin{aligned}\mathbf{g}^{(t)} &= \nabla E(\mathbf{w}^{(t)}) \\ \mathbf{m}_t &= \rho_1 \mathbf{m}_{t-1} + (1 - \rho_1) \mathbf{g}^{(t)} \\ \mathbf{v}_t &= \max\{\rho_2 \mathbf{v}_{t-1}, |\mathbf{g}^{(t)}|\} \\ \Delta \mathbf{w}^{(t)} &= -\frac{\alpha}{1 - \rho_1^t} \frac{\mathbf{m}_t}{\mathbf{v}_t} \\ \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \Delta \mathbf{w}^{(t)}\end{aligned}$$

- Nadam (2016)

ネステロフの加速法を Adam に取り入れたものが Nadam

$$\begin{aligned}
 \mathbf{g}^{(t)} &= \nabla E(\mathbf{w}^{(t)}) \\
 \mathbf{m}_t &= \mu_t \mathbf{m}_{t-1} + (1 - \mu_t) \mathbf{g}^{(t)} \\
 \mathbf{v}_t &= \rho \mathbf{v}_{t-1} + (1 - \rho) (\mathbf{g}^{(t)})^2 \\
 \hat{\mathbf{m}}_t &= \frac{\mu_{t+1}}{1 - \prod_{i=1}^{t+1} \mu_i} \mathbf{m}_t + \frac{1 - \mu_t}{1 - \prod_{i=1}^t \mu_i} \mathbf{g}_t \\
 \hat{\mathbf{v}}_t &= \frac{\rho}{1 - \rho^t} \mathbf{v}_t \\
 \Delta \mathbf{w}^{(t)} &= - \frac{\alpha_t}{\sqrt{\hat{\mathbf{v}}_t} + \varepsilon} \hat{\mathbf{m}}_t \\
 \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \Delta \mathbf{w}^{(t)}
 \end{aligned}$$

- Eve (2016)

相対的変化が大きければ学習率を削減し、小さければ上昇させることでフラットエリアで学習を高速化できるよう Adam を改良したもの。

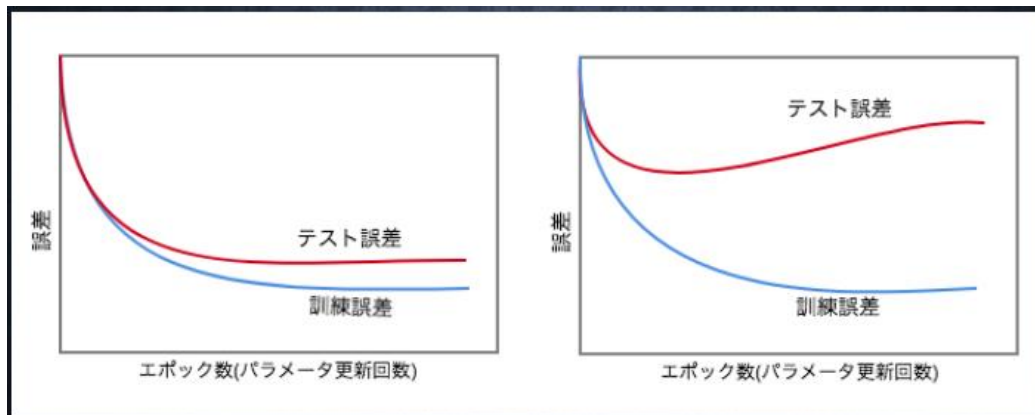
$$\begin{aligned}
 \mathbf{g}^{(t)} &= \nabla E(\mathbf{w}^{(t)}) \\
 \mathbf{m}_t &= \rho_1 \mathbf{m}_{t-1} + (1 - \rho_1) \mathbf{g}^{(t)} \\
 \mathbf{v}_t &= \rho_2 \mathbf{v}_{t-1} + (1 - \rho_2) (\mathbf{g}^{(t)})^2 \\
 \hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1 - \rho_1^t} \\
 \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1 - \rho_2^t} \\
 \delta_t &= \begin{cases} k+1 & (E(\mathbf{w}^{(t-1)}) \geq \hat{E}_{t-2}) \\ 1/(k+1) & (E(\mathbf{w}^{(t-1)}) < \hat{E}_{t-2}) \end{cases}, \Delta_t = \begin{cases} K+1 & (E(\mathbf{w}^{(t-1)}) \geq \hat{E}_{t-2}) \\ 1/(k+1) & (E(\mathbf{w}^{(t-1)}) < \hat{E}_{t-2}) \end{cases} \\
 c_t &= \min \left\{ \max \left\{ \delta_t, \frac{E(\mathbf{w}^{(t-1)})}{\hat{E}_{t-2}} \right\}, \Delta_t \right\} \\
 \hat{E}_{t-1} &= c_t \hat{E}_{t-2} \\
 r_t &= \frac{|\hat{E}_{t-1} - \hat{E}_{t-2}|}{\min\{\hat{E}_{t-1}, \hat{E}_{t-2}\}} \\
 d_t &= \rho_3 d_{t-1} + (1 - \rho_3) r_t \\
 \Delta \mathbf{w}^{(t)} &= - \frac{\eta}{d_t \sqrt{\hat{\mathbf{v}}_t} + \varepsilon} \hat{\mathbf{m}}_t \\
 \mathbf{w}^{(t+1)} &= \mathbf{w}^{(t)} + \Delta \mathbf{w}^{(t)}
 \end{aligned}$$

などがある。意外と多くのアルゴリズムがある印象。

Section3：過学習

テスト誤差と訓練誤差とで学習曲線が乖離すること

特定の訓練サンプルに対して、特化して学習してしまった状態を指す。(下図 右側)



過学習の原因について

1つはパラメータ数が多いことが原因となる。(必要以上に高次元モデルとなっている)

その他に入力データが少ない。

この結果、ニューラルネットワークの表現力は高いが、データが少ないため過剰に適合してしまう。

正則化による過学習の抑制

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抑制する

- L1 正則化、L2 正則化

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。

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

:誤差関数に、pノルムを加える

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

:pノルムの計算

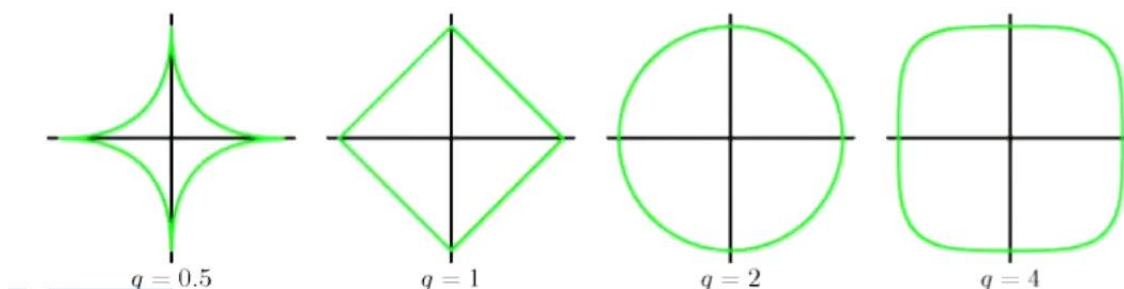
P=1 の場合を L1 ノルム (ラッソ回帰)

P=2 の場合を L2 ノルム (リッジ回帰) と呼び、主に使われる。

重みの更新では、下記計算を行う。

$$\|\mathbf{W}^{(1)}\|_p = \left(|\mathbf{W}_1^{(1)}|^p + \dots + |\mathbf{W}_n^{(1)}|^p \right)^{\frac{1}{p}}$$

正則化を行う際のイメージは下記となる。



上記正則化の枠と誤差関数の接点に学習するため、L1 ノルムでは不要な重みが 0 にすることが可能。L2 ノルムにおいても 0 に近づけることができる。

- ドロップアウト

ニューラルネットワークのノード数が多いほど自由度の高い関数となる。

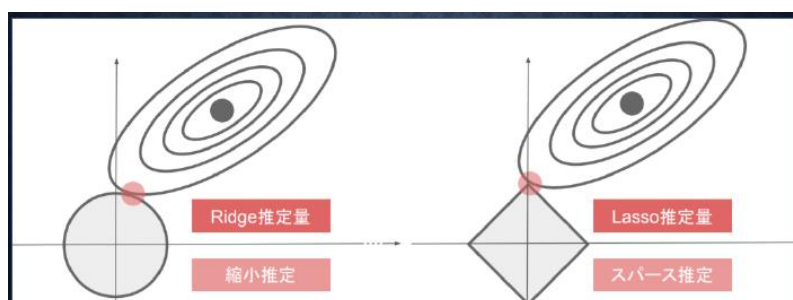
そこで、ランダムにノードを削除して学習させることをドロップアウトと呼ぶ。

- 確認テスト

[機械学習で使われる線形モデルの正則化は、モデルの重みを制限することで可能となる。その線形モデルの正則化手法の中で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化項を加える

[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。]



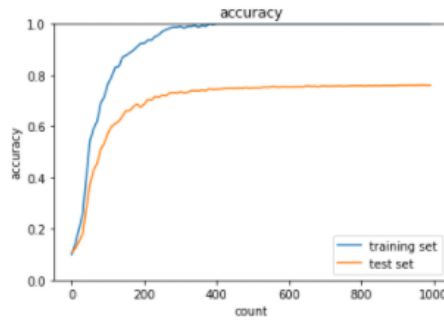
解答：右図

また、左側は L2 正則（ラッソ回帰）となる。

L2 正則では誤差関数と円の接点となるため、0 になることは少ないと考えられる。

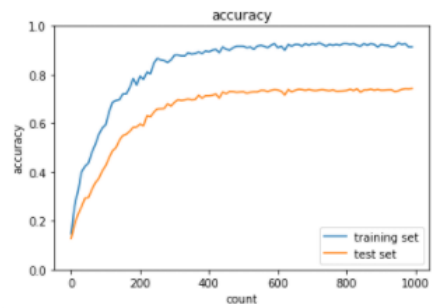
・実装演習

2_5_overfitting.ipynb



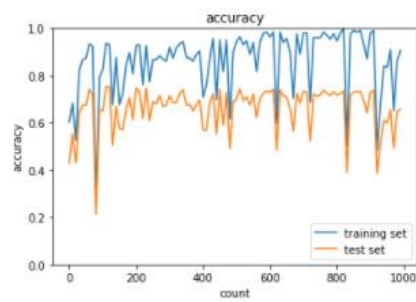
Train データは 100% となっており過学習気味。

L2 ノルム



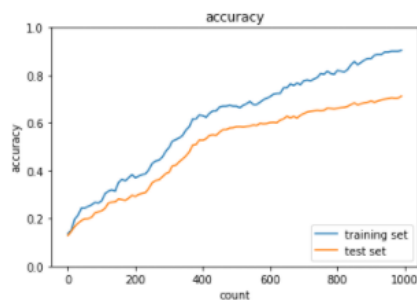
正則化で Train データへの適合が低下。

L1 ノルム



正則化でかなり不安定に。

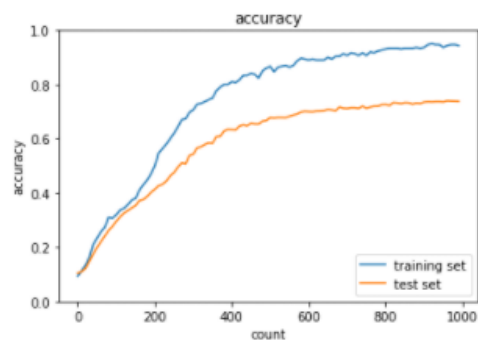
Dropout



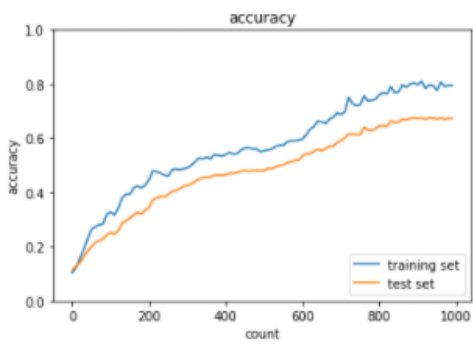
Train/Test とともに順調に学習。

Dropout + L1

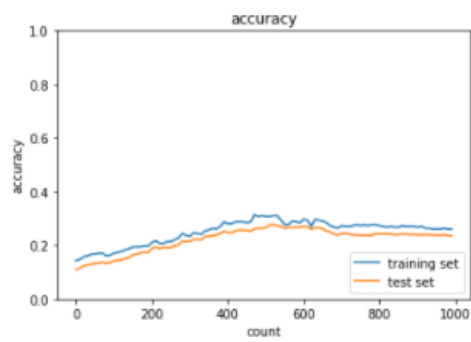
dropout_ratio = 0.08



dropout_ratio = 0.1



dropout_ratio = 0.2



dropout を 8%→10%→20%と増やしたが
少し増やすと影響が大きい。

・その他

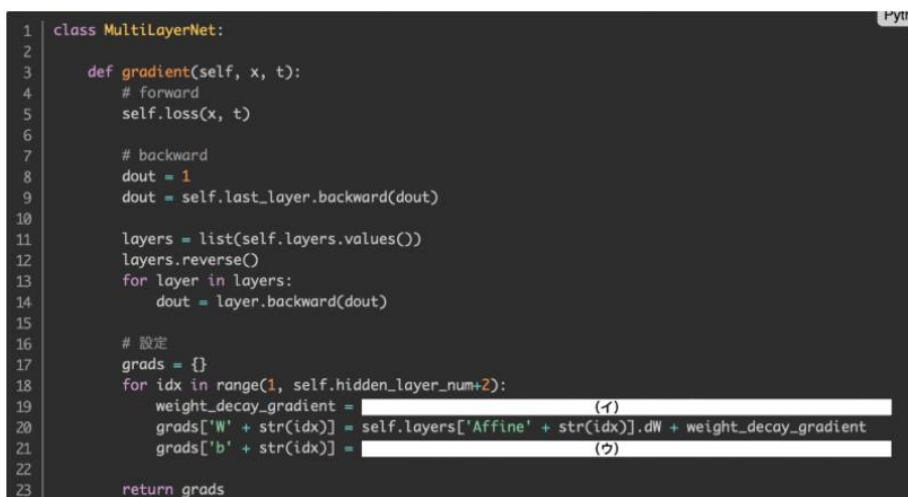
練習用：E 資格受験し確認点テストより

過学習を抑制するために用いられる手法として荷重減衰 (weight decay) と呼ばれる手法がある。これは、学習の過程において、大きな重みを持つことにペナルティを課すことで、過学習を抑制しようとするのである。

問題

以下のプログラム【図1】はあるニューラルネットワークの実装の一部である。gradient関数は各層の重みの勾配を求める関数である。空欄（イ）では各層の重みに対して、L2正則化項の微分値を足し合わせている。【図1】空欄（イ）に当てはまる選択肢を選べ。

1. `0.5 * self.weight_decay_lambda * (self.layers['Affine' + str(idx)].W **2)`
2. `self.weight_decay_lambda * self.layers['Affine' + str(idx)].W`
3. `self.weight_decay_lambda + self.layers['Affine' + str(idx)].W`
4. `0.5 * self.weight_decay_lambda * self.layers['Affine' + str(idx)].W`



```
1 class MultiLayerNet:
2
3     def gradient(self, x, t):
4         # forward
5         self.loss(x, t)
6
7         # backward
8         dout = 1
9         dout = self.last_layer.backward(dout)
10
11         layers = list(self.layers.values())
12         layers.reverse()
13         for layer in layers:
14             dout = layer.backward(dout)
15
16         # 設定
17         grads = {}
18         for idx in range(1, self.hidden_layer_num+2):
19             weight_decay_gradient = (イ)
20             grads['W' + str(idx)] = self.layers['Affine' + str(idx)].dW + weight_decay_gradient
21             grads['b' + str(idx)] = (ウ)
22
23         return grads
```

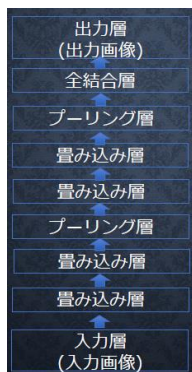
重み W の $L2$ ノルムを損失関数に加算を行う。

重み W とすれば、 $L2$ ノルムの Weight decay は正則化項 $\frac{1}{2\lambda W^2}$ の微分 λW を加算する。

2. `self.weight_decay_lambda * self.layers['Affine' + str(idx)].W`

Section4：畳み込みニューラルネットワークの概念

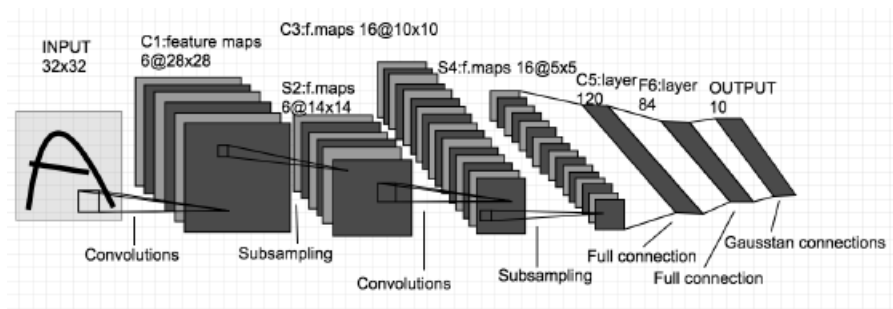
畳み込みニューラルネットワーク（CNN）の構成例は下記となる。



畳み込みとプーリング層、全結合層を用いることによって
CNN では次元間でつながりのあるデータを扱える。

LeNet について

32x32 の入力から、10 個の出力を出すネットワークとなる。



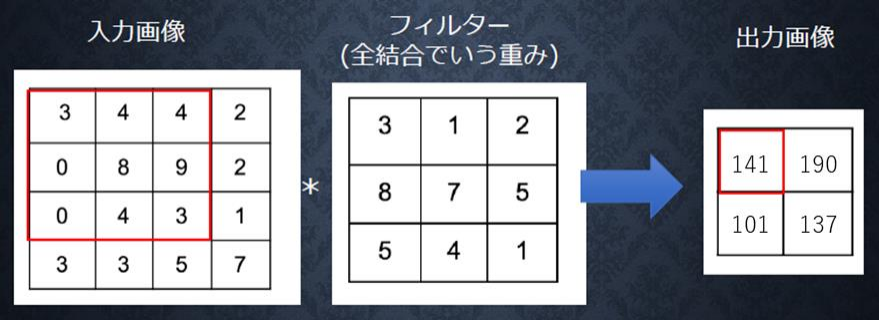
- ① 入力：(32, 32) 1024 個
- ② 特微量マップ (28, 28, 6) 4704 個
- ③ 特微量マップ (14, 14, 6) 1176 個
- ④ 特微量マップ (10, 10, 16) 1600 個
- ⑤ 特微量マップ (5, 5, 16) 400 個
- ⑥ 全結合層 (120) 120 個
- ⑦ 全結合層 (84) 84 個
- ⑧ 全結合層(10)

畳み込み層の全体像

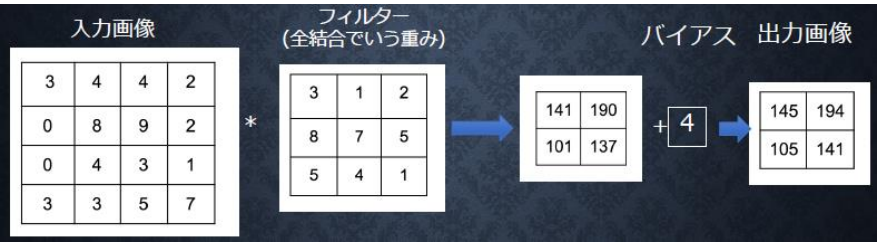
入力値*フィルターによって出力

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。

結論: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。

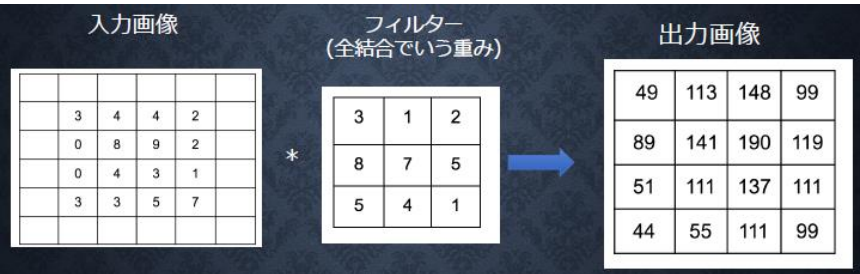


バイアス



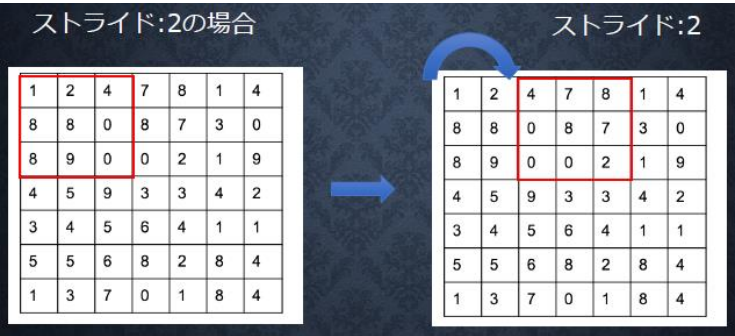
パディング

入力データの外側に指定分のデータ拡張を行うことを指す。
入力データ外周の情報量やデータサイズの点で利用されている。



ストライド

フィルターを移動するステップ数を指している。



全結合層

重みと入力かけた通常のニューラルネットワーク

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される。

そのため、情報の関連性を持たせるために畳み込み層が活用されるようになった。

・実装演習

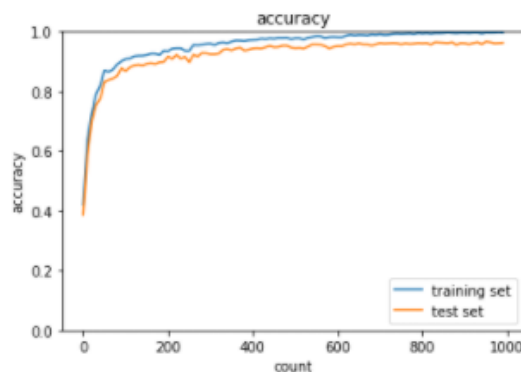
2_6_simple_convolution_network.ipynb

```
1 # im2colの処理確認
2 input_data = np.random.rand(2, 1, 4)*100//1 # number, channel, height, widthを表す
3 print('==========\n', input_data)
4 print('=====  
5 filter_h = 3
6 filter_w = 3
7 stride = 1
8 pad = 0
9 col = im2col(input_data, filter_h=filter_h, filter_w=filter_w, stride=stride, pad=pad)
10 print('=====  
11 print('=====')
```

```
=====  
[[[[[51. 70. 12. 52.]  
[12. 36. 1. 47.]  
[82. 61. 73. 92.]  
[ 3. 58. 27. 20.]]]  
  
[[[28. 41. 52. 7.]  
[30. 3. 24. 17.]  
[39. 56. 1. 64.]  
[50. 51. 74. 62.]]]]]  
=====
```

```
=====  
[[51. 70. 12. 12. 36. 1. 82. 61. 73.]  
[70. 12. 52. 36. 1. 47. 61. 73. 92.]  
[12. 36. 1. 82. 61. 73. 3. 58. 27.]  
[36. 1. 47. 61. 73. 92. 58. 27. 20.]  
[28. 41. 52. 30. 3. 24. 39. 56. 1.]  
[41. 52. 7. 3. 24. 17. 56. 1. 64.]  
[30. 3. 24. 39. 56. 1. 50. 51. 74.]  
[ 3. 24. 17. 56. 1. 64. 51. 74. 62.]]  
=====
```

Generation: 1000. 正答率(トレーニング) = 0.9956
: 1000. 正答率(テスト) = 0.962



CNN を用いたニューラルネットワークでは汎化性能が高くなっていることがわかった。

これは、講義内でもあったプーリング層を利用している効果なのだろうか。

・その他

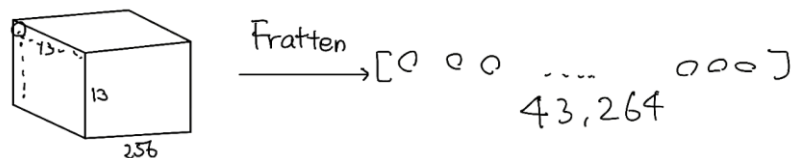
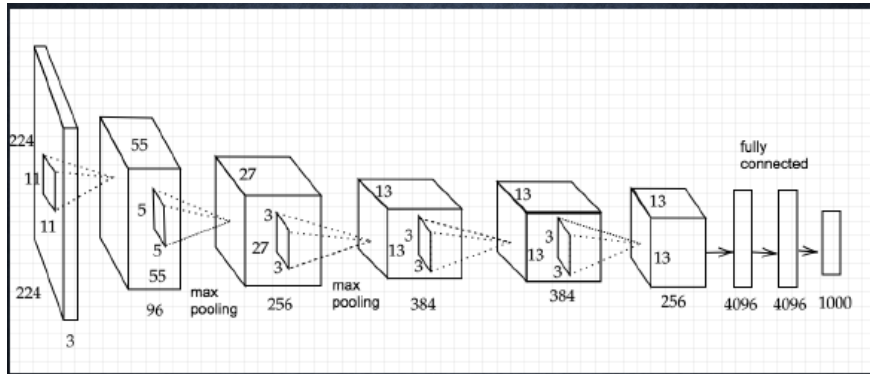
[サイズ 6×6 の入力画像を、サイズ 2×2 のフィルタで畳み込んだ時の出力画像のサイズを答えよ。なおストライドとパディングは 1 とする。]

解答: 5×5

Section5：最新の CNN

AlexNet のモデル

5 層の畳み込み層及びプーリング層など、それに続く 3 層の全結合層からなる。



Global Max Pooling 最大値をとる

Global Avg Pooling 平均をとる

なぜか Fratten よりも、Pooling のほうが画像処理ではうまくいく。

過学習を防ぐ施策

- ・サイズ 4096 の全結合層の出力にドロップアウトを使用している。

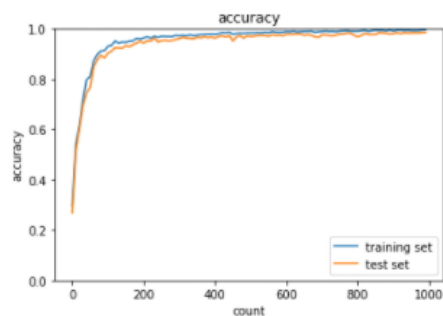
- ・実装演習

2_8_deep_convolution_net.ipynb

今までの学習に比べ学習時間は非常に長かったが、正答率は良好な結果となる。

simple_convolution では 96%の結果が、今回 98%強まで高まっている。

Generation: 1000. 正答率(トレーニング) = 0.9958
: 1000. 正答率(テスト) = 0.985



時間はかかる

✓ 1時間 11分 32秒 完了時間: 18:20