平成 **22** 年度　特別研究報告書

SOM 特徴マップ初期値の改善による可視化安定性の向上

龍谷大学　理工学部　情報メディア学科

学籍番号 **T070470**　百井 伸次

指導教員　三好 力 教授

内容梗概

高次元データに対する視覚的理解の手助けのために自己組織化マップという技術が存在する。自己組織化マップは距離で表現されるものなので、特徴マップの初期値は非常に重要な要素である。しかし既存技術ではその特徴マップの初期値は乱数によって決められるため、同じデータを用いても毎回違う出力結果になってしまい、その安定性に欠けるという問題点がある。また、学習データを解析したものを基準とする初期化手法が提案されているが、自己組織化マップには解析能力が十分あると考えられるので、学習データに手を加えすぎると本来の目的を損ないかねない。そこで本研究では学習データを参考にすることを少なくした上で、自己組織化マップの出力結果の安定性向上を目的とした特徴マップの初期化手法を提案し、既存手法との比較実験により出力結果の検証を行った。

比較実験は初期化手法が既存のものと提案手法を同じ環境で繰り返し学習実験を行い、出力結果を視覚的に、また一定のデータに対する勝利ノード座標を標準偏差で評価した。結果、既存手法と比較して視覚的に安定していると判断でき、また標準偏差からも定量的に安定性が向上しているということがわかった。

目次

第一章　はじめに **2**
1.1　研究背景 2
1.2　自己組織化マップとは 3
1.3　自己組織化マップの基本的なアルゴリズム 3

第二章　既存技術の問題点 **7**
2.1　参照ベクトルの初期値について 7
2.2　入力データを分析したものを参考にした場合について 8
2.3　入力データの分析を抑えて初期化を行う既存手法 8

第三章　提案手法 **9**
3.1　提案手法（１） 9
3.1.1.　乱数の範囲の決定 9
3.1.2.　参照ベクトルのソート 9
3.1.3.　提案手法のアルゴリズム 9
3.2　実験方法（１） 11
3.2.1.　実験パラメータ（１） 11
3.2.2.　実験結果（１） 12
3.3　別視点からの安定性比較実験（１） 13
3.4　考察（１） 17

第四章　提案手法（２） **18**
4.1　入力データの参照数の決定 18
4.2　実験方法（２） 19
4.2.1.　実験パラメータ（１） 19
4.2.2.　実験結果（２） 20
4.3　別視点からの安定性比較実験（２） 21
4.4　考察（２） 25

第五章　おわりに **26**

謝辞 **27**

参考文献 **28**

付録

第一章　はじめに

1.1　研究背景

データマイニングの一手法として健康診断や事業戦略管理、検索システムなどに使われている自己組織化マップという技術が存在する。

世の中に複雑な情報が数多く存在しているが、人間はそのような高次元の情報を瞬時に分析、判断することはできない。自己組織化マップは多次元データ間の類似度を二次元特徴マップ上での距離で表現することによって、そのような多くの要素を持ったデータ同士の関係について、人間が視覚的に判りやすくなるよう表す技術である。

自己組織化マップはデータの視覚化の他にも、データの分類・要約分析を得意とする。データの分類に関しては、出力マップ上で入力データ内で似た特徴を持つものは近くに、全く異なる特徴を持つものは遠くにマッピングされるため、データが似ているかどうか判断しやすいためである。データの要約分析に関しては、多量のデータの類似度を視覚化することによって視覚化前では気付くことのできなかったデータ間の情報を得ることができるためである。

自己組織化マップによって学習を行う際、重要となってくる点が特徴マップの初期化である。それは入力データの値と特徴マップが持つ各ベクトルが持つ値とのユークリッド距離を計算・比較を行うからであり、最初の学習結果は特徴マップが持つ各ベクトルの初期値に依存するからである。

既存技術ではその特徴マップが持つ各ベクトルの初期値を乱数で決定しているが、それでは結果出力のたびに違う結果が表れてしまう。同じ入力データでも違う出力結果が表れてしまうよりは同じ入力データなら似た出力結果が表れた方が、自己組織化マップの目的の一つである人間の視覚的理解の助けに繋がると考えられる。

これまでに安定した出力結果を得るために、学習データの値を解析したものを基準として特徴マップを初期化する手法が提案されているが、あまり学習データに手を加えすぎると自己組織化マップそのものが持つデータ解析能力の意味が薄れてしまう。また、学習データ数や学習データの次元数が多い場合には初期化時に膨大な計算量を割くことになってしまう。

ここで本稿では、学習データを参考にすることを出来るだけ少なくし、かつ自己組織化マップによる結果出力を安定したものにするため、学習データから最小値・最大値のみ取得し、その範囲で生成した乱数を特徴マップ上のベクトルに与え、そのベクトルをソートをすることによって初期化時の特徴マップに傾きを持たせる手法を提案する。また、最小値・最大値の取得方法、ソートの基準となる値についても比較・検討を行う。

1.2　自己組織化マップとは

　自己組織化マップ(Self Organization Mapping:通称 SOM)とは、競合学習型ニューラルネットワークの一種であり、入力層と出力競合層の 2 層から成っている。自己組織化マップは 1980 年代にコホーネンによって開発され、多次元データの分類、解析に効果的な技術として知られている。

　自己組織化マップの特徴は、n 次元のベクトル集団を学習することにより 2 次元のマップにそれらのベクトルの関係を写像することができることである。似ているベクトルは 2 次元マップ上の近い位置に配置され、似ていないベクトルは遠い位置に配置される。汎用性に優れており、認識、予測、分類など様々な応用が可能であることから、今後の活躍が期待できる。

1.3　自己組織化マップの基本的なアルゴリズム

コホーネンは生物の神経細胞、主に脳の情報処理の仕方を以下の簡単な式に整理した。

$$m_i(t+1)=m_i(t)+h_{ci}(t)[x(t)-m_i(t)]. \tag{1.1}$$

この式は、次のような意味である。いま神経細胞(ノード)i が時刻 t で処理している情報処理能力を $m_i(t)$ とするとき、外部から入力信号 $x(t)$ が入ってきたとする。細胞はこの入力信号を学習して次の時刻には入力信号により近い情報処理能力 $m_i(t+1)$ を持つようになる。このとき $x(t)$ が n 次元の入力ベクトルである場合、参照ベクトルとも呼ばれる $m_i(t)$ もまた同じ n 次元の要素を持つ。そして、 $h_{ci}(t)$ は学習率係数を含めた近傍関数を意味する。なお、 $t=0,1,2\cdots$ は離散時間座標である。出力競合層のベクトルは参照ベクトル $m_i(t)$ で表され、入力層の次元に合わせて n 個の要素を持っている。図○に示すように、視覚的に出力を見るために、普通は 2 次元に配列されている。

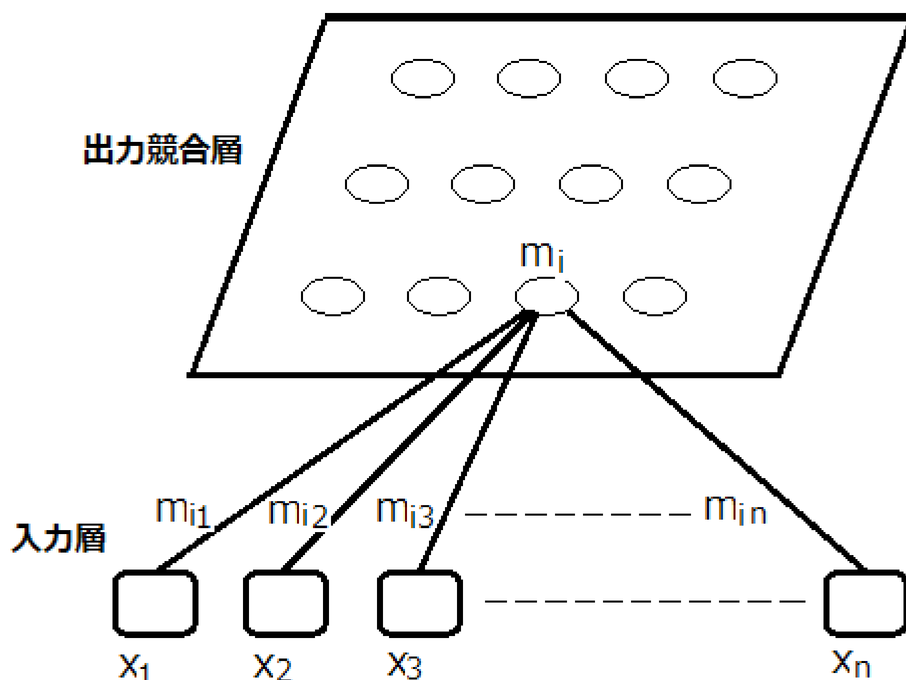
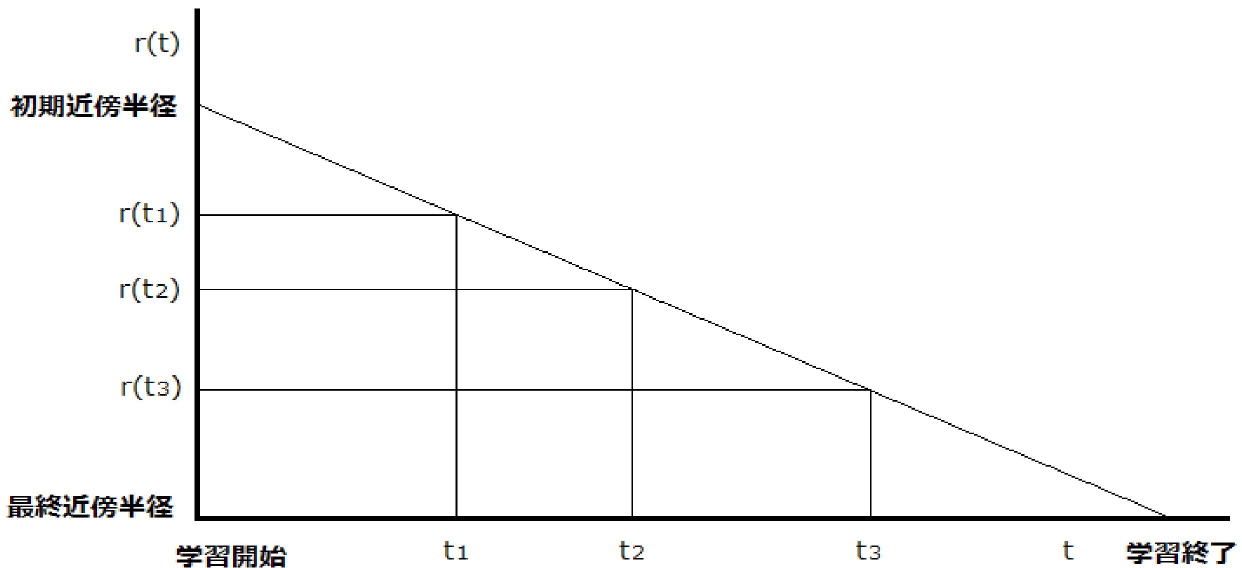


図 1 .自己組織化マップの構造

自己組織化マップの学習は次のように行われる。入力ベクトル $x(t)$ はある測度、例えばユークリッド距離 $|x(t)-m_i(t)|$ を最小にするノード i を探し、それに添え字 C をつける。

$$|x(t)-m_c(t)|=min|x(t)-m_i(t)|. \tag{1.2}$$

式 (1.2)で決められた参照ベクトル $m_c(t)$ を持つノードを勝者ノードと呼ぶ。式(1.1)および式(1.2)での学習の状態について下図〇で簡単に説明する。まず入力ベクトルが提示されると、その入力ベクトルに一番近いノードが勝者ノードとなる。そして勝者ノードの周りに囲った正方形の図では、円になっている領域を近傍領域と定義する。その形は正方形でもよい。基本のノードの配列を六角形の形にとれば六角形近傍となる。近傍内のすべてのノードは入力ベクトルを学習し、式(1.1)に従って入力ベクトルの方向へ少し近づく。この学習を繰り返し行うが、このとき近傍領域の範囲は最初広くとっておき、学習とともにその領域を狭めていく。



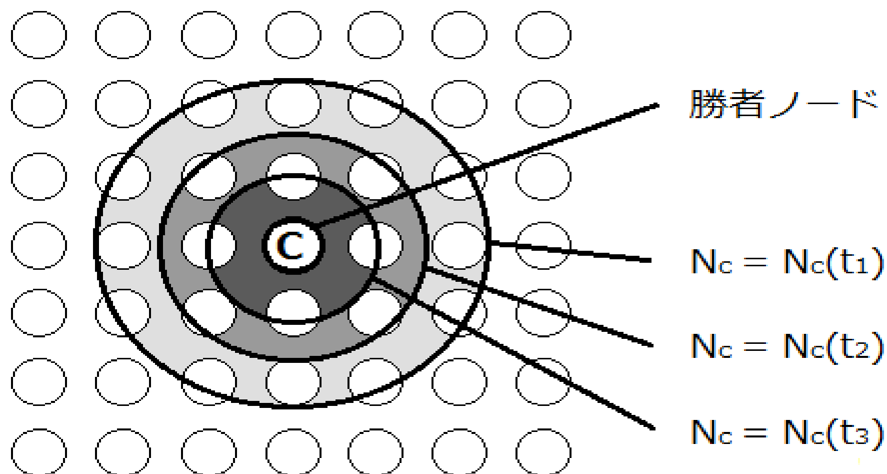


図２.自己組織化マップの学習の繰り返しにおける近傍領域の変化

式(1.1)において、 $h_{ci}(t)$ は、学習率係数 $\alpha(t)$ と近傍関数 $h(d,t)$ により次のように表現できる。

$$h_{ci}(t)=\alpha(t)*h(d,t). \tag{1.3}$$

近傍関数は、学習される近傍領域を指定する関数である。近傍領域は、出力競合層の勝利ノードの近傍を意味し、学習によって学習ベクトルが更新される領域である。学習を始めるときには、近傍領域をその範囲を大きくとり、学習が進むにつれて徐々にその領域を狭める。近傍領域を減少させる関数には、線形型とステップ型とがある。

ここでは、よく使用されている、ガウス型近傍関数について説明する。

$$h(d,t)=\exp(-d^2/k(r(t))^2). \tag{1.4}$$

ここで、 d は式(1.5)で表せる勝者ノードから参照ベクトルまでの距離、 $k(r(t))$ は学習時間 t のときの近傍領域の最大距離で、ガウス関数の波幅の係数である。

$$d^2=(x-a_i)^2+(y-b_i)^2. \tag{1.5}$$

ここで、 (a_i, b_i) は勝者ノードの位置、 (x, y) は半径 $r(t)$ により成る円形領域の内側にあるノードの位置を示す。

近傍半径 $r(t)$ は、学習のはじめの段階では大きく、学習が進むにつれて小さくなってゆく。

半径 $r(t)$ の円領域に含まれる範囲のノード、参照ベクトルは、指数関数で決まる重みを持って学習される。すなわち、勝者ノードに近いほど、その類似性が大きくなるように学習され、勝者ノードから遠ざかるほど、その類似性が小さくなるように学習される。そして半径 $r(t)$ の範囲外では学習されないことになる。

したがって、式(1.1)によって学習している間、近傍領域 N_C 外のノードに関しては、 $h_{ci}(t)=\alpha(t)*h(d,t)$ で、 N_C 外のノードに関しては、 $h_{ci}(t)=0$ である。つまり、近傍の外側の領域については学習されない。

以上により、近傍関数は次式で表される。

$$\begin{aligned} &h_{ci}(t)=\alpha(t)*h(d,t) && (i\subset N_C) \\ &h_{ci}(t)=0 && (\text{それ以外}). \end{aligned} \tag{1.6}$$

このとき、 $\alpha(t)$ の値を学習率係数と呼び、 $0<\alpha(t)<1$ の値を持つ。 $\alpha(t)$ と N_C の大きさは両方とも学習時間が経つにつれて普通、単調減少させる。 $\alpha(t)$ は例えば次の式で定義しても良い。

$$\alpha(t)=\alpha_0(1-t/T). \tag{1.7}$$

ここで、 α_0 は α の初期値であり、普通0.2〜0.5の値を選ぶ。Tは行われるべき学習での予定された全更新学習回数である。ただし、式(1.1)中、比例係数の $\alpha(t)$ は、学習のはじめでは大きな値をとるようにし、学習が進んでくるとだんだん小さい値になるようにする。また、近傍領域 $N_C=N_C(t)$ も式(1.6)と同様に減らしていってもよい。つまり、

$$N_C(t)=N_C(0)(1-t/T). \tag{1.8}$$

ここで、 $N_C(0)$ は初期値である。

コホーネンの自己組織化マップアルゴリズムを整理すると、以下のようになる。

1. 出力層にノードを配置し、それぞれの持つ参照ベクトルを乱数で初期化する。
2. 入力ベクトルに最も近い(似ている)競合層での参照ベクトルを探し、これを勝者ノードとする。
3. この勝者ノード及び近傍内のノードを式(1.1)に従って更新する。また、学習が進むにつれて近傍領域を狭め、学習率係数の値も例えば式(1.4)のようにして減らしていく。

自己組織化マップの学習を行うために学習パラメータの設定をする必要がある。ここに主要な学習パラメータとその説明を示す。

学習回数　　：競合層に対して学習を何回行うかを指定する。
学習係数　　：1回の学習でノードをどれだけ更新するかを指定する。
近傍領域　　：どれだけの範囲に学習の影響を与えるか初期範囲を指定する。
マップサイズ：出力競合層のノードをX,Y軸にいくつ並べるかを指定する。

第二章　既存技術の問題点

2.1　参照ベクトルの初期値について

1.1 や 1.3 で述べたように、特徴マップの各ベクトル（以下参照ベクトル）が持つ値は、既存技術では乱数で初期化されている。しかし、ただ単に乱数で初期化した場合では、自己組織化マップの学習時、主に初回の学習において大きな影響を及ぼす。以下に単に乱数で初期化した場合で起こりうる問題点の例を挙げる。

・同じ参照データでも勝利ノードの座標が大きく変わる

参照ベクトルが持つ値を乱数で決定すると、学習の度に各参照ベクトルが持つ値は変わる。そうなるとある学習時に参照した入力ベクトルと、別の学習時に参照した入力ベクトルに対する勝利ノードの座標が大きく違ってくる場合が多い。その例を図 3 に示す。

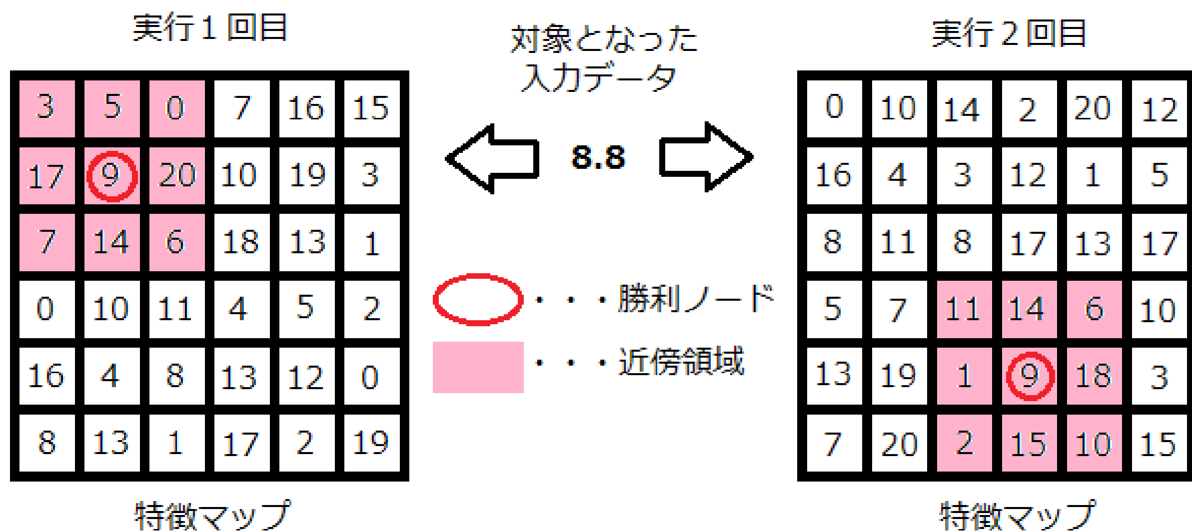


図 3 .乱数初期化における勝利ノード座標のばらつきの例

例では特徴マップが 6×6、入力ベクトルが 8.8、近傍領域が周囲 1 マスとなっている。このように参照ベクトルの値が完全にランダムであると、特徴マップ上での入力データの扱いがたとえ同じ値であったとしても自己組織化マップを実行する度に同様に完全にランダムになってしまう。また、勝利ノードの周辺のベクトル、つまり近傍領域内のベクトルも値はランダムなため、影響を受けるベクトルも自己組織化マップを実行する度に大きく変わってしまう。

さらに、一回入力データに対する勝利ノードが決定してしまうと、再度その入力データが参照された場合やその入力データに似た入力データが参照された場合でも、その勝利ノードやその近辺のベクトルが勝利ノードに選ばれる可能性が高くなる。したがって、初期値が自己組織化マップ実行毎に大きく変わってくると、出力結果も大きく変わっていくので、初期値が結果に及ぼす影響は強いと考えられる。

2.2　入力データを分析したものを参考にした場合について

初期値を完全に乱数で決定する方法に対して、初期値を入力データから計算した値で決定する方法が提案されている。例えば、Mu-Chun su らによって提案された手法[3)]では、入力データ内での距離が最も遠い 4 つを抽出し、それを特徴マップの四隅に割り当て重みとし、さらに両端の重みから最も遠いデータを対辺の重みとして割り当て、それをマップが埋まるまで時計回りに繰り返した結果できる特徴マップを初期値とする方法が提案されている。この例のように、入力データを解析して初期値を決定する方法について、以下のような問題点が挙げられる。

・自己組織化マップとの目的の類似

本来の目的として、入力データについてどのような関連性・性質があるかどうかを分析するために、自己組織化マップが用いられる。参照ベクトルの初期値を決めるために本来分析の対象となる入力ベクトルを別の手法で先に分析してしまうということは、程度によっては本末転倒になりかねない。

・対象となる入力データの値の数に計算量が依存する

対象となる入力データの数、及び次元数がが多ければ多いほど、分析のための計算量や計算回数が多くなっていく。さらに分析した値を特徴マップに割り当てる際も、並び順なども考慮したアルゴリズムを別途組む必要があり、入力データの分析と特徴マップの割り当てとで多くのステップが必要になってしまう。

2.3　入力データの分析を抑えて初期化を行う既存手法

三好らの研究では自己組織化マップの学習能力を十分利用するため、多くの計算コストをかけることなく、初期値をランダムに決定する方法を改善することによって学習速度の高速化を行う方法[4)]が提案されている。この方法は、初期化時点で学習するデータを参考に特徴マップのノード交換を行うことによって、入力ベクトル空間と特徴マップ上の位置関連付けを行うものである。この方法を導入することで全ノードの平均移動距離が短縮され、学習速度の高速化が確認されている。しかし、自己組織化マップの学習能力を生かす点と初期化手法に手を加える点については本稿と合致はしているが、本稿の出力結果の安定性を求める目的については十分検討されているとは言えない。

第三章　提案手法

ここでは前章で示した問題点に対して提案手法を挙げる。その前に前章以前でも述べたが、提案手法を挙げるにあたって重視することを挙げる。

・学習データの分析を少なく抑える
・参照ベクトルの初期値を決めるにあたって、なるべく計算量を抑える

3.1　提案手法（１）

アルゴリズムを後述するが、今提案手法では学習データの分析を少なく抑えるために、特徴マップ上のベクトルを生成する際に基準となる値を学習データに対して大掛かりな計算をせずに決定した。しかし、単にこれだけでは安定性の向上については見込めないため、特徴マップの値の決定後に各ノードのソートを行うことによって特徴マップに傾きを持たせることで安定性の向上を図った。この手法は以下に示す二点を特徴としている。

3.1.1　乱数の範囲の限定

学習データ内の最小値・最大値のみ取得し、その範囲で乱数を生成するということを考える。そしてその生成した乱数を参照ベクトルの初期値として暫定的に割り振る。単に乱数で参照ベクトルを決定するより、最小値・最大値を取得することで、参照ベクトルの値の範囲が学習データの持つ値の範囲内全てに対応したものになるという利点がある。また最小値・最大値を取得するだけでよいため、学習データに対して大掛かりな計算をする必要はない。

3.1.2　参照ベクトルのソート

次に、暫定的に初期値として割り振った参照ベクトルが持つ値の平均値を計算し、その平均値を基に特徴マップをソートする。これを行うことで参照ベクトルが昇順に並ぶことになり、特徴マップに傾きが表れる。特徴マップに傾きを持たせることで、前章で述べた同データに対する勝利ノードの位置の振れ幅を小さく出来、結果的に最終出力結果も安定したものになると考えられる。

3.1.3　提案手法のアルゴリズム

以下に3.1及び3.2までの提案手法のアルゴリズムを示す。

1. 入力データ全てを参照する
2. 参照した中から最小値・最大値を取得する
3. 最小値・最大値の範囲で乱数を生成する
4. 生成した乱数を参照ベクトルに暫定的に初期値として割り当てる
5. 参照ベクトルの次元数分の値の平均値を求める
6. 平均値を基に、特徴マップの縦方向に対して参照ベクトルを昇順に、クイックソートを行う

7. 平均値を基に、特徴マップの横方向に対して参照ベクトルを昇順に、クイックソートを行う
8. ソートによって出来た特徴マップを初期値とし、学習を行う

3.2　実験方法（１）

今回の実験にあたって、入力データと特徴マップに特に手を加えていない既存手法の自己組織化マップのプログラムを作成した。加えて、前章の提案手法を導入した自己組織化マップのプログラムを作成した。

それぞれのプログラムの概要としては、指定した data ファイルを入力データとして読み込み、自己組織化マップによって得られた値を data ファイル化し、gnuplot によって 3 次元グラフ化されたものを上空視点で表示することでマップ化するというものになっている。

また、得られる出力結果がわかりやすいように、値が二極化した入力データを用いる。以下に二極化した入力データのイメージを示す。

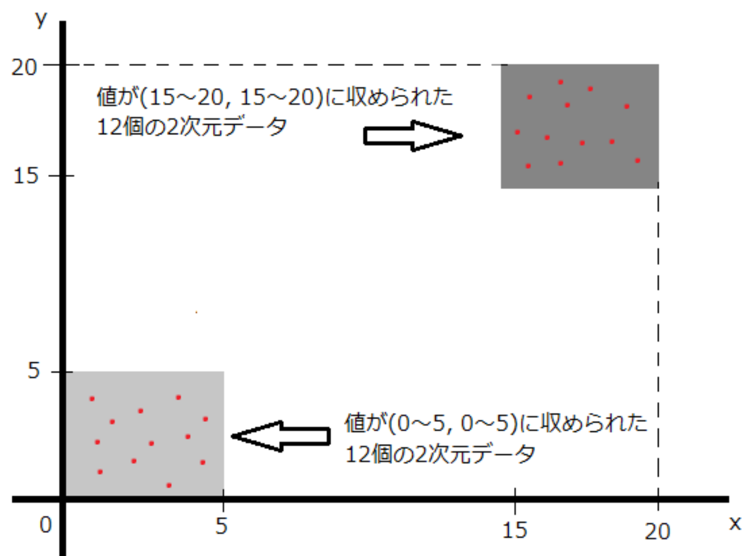


図４.二極化した入力データのイメージ

以上の二つの自己組織化マップを複数回実行し、出力結果が安定したものであるかどうか比較・検討を行う。

3.2.1　実験パラメータ（１）

以下のパラメータで自己組織化マップを実行する。

- 実験回数　：5 回
- 入力データ：2 次元×24 個のデータ、値は二極化されている
- 特徴マップ：30×30 マス
- 学習回数　：1000 回
- 近傍領域　：20（学習回数 50 毎に 1 減少）
- 学習率係数：0.01×近傍領域

3.2.2　実験結果（１）

以下に入力データと特徴マップに特に手を加えていない既存手法の自己組織化マップと提案手法である乱数の範囲を指定した後に次元値の平均で特徴マップをソートした自己組織化マップの出力結果を図５に示す。図５の左が既存手法の自己組織化マップ、右が提案手法の自己組織化マップである。

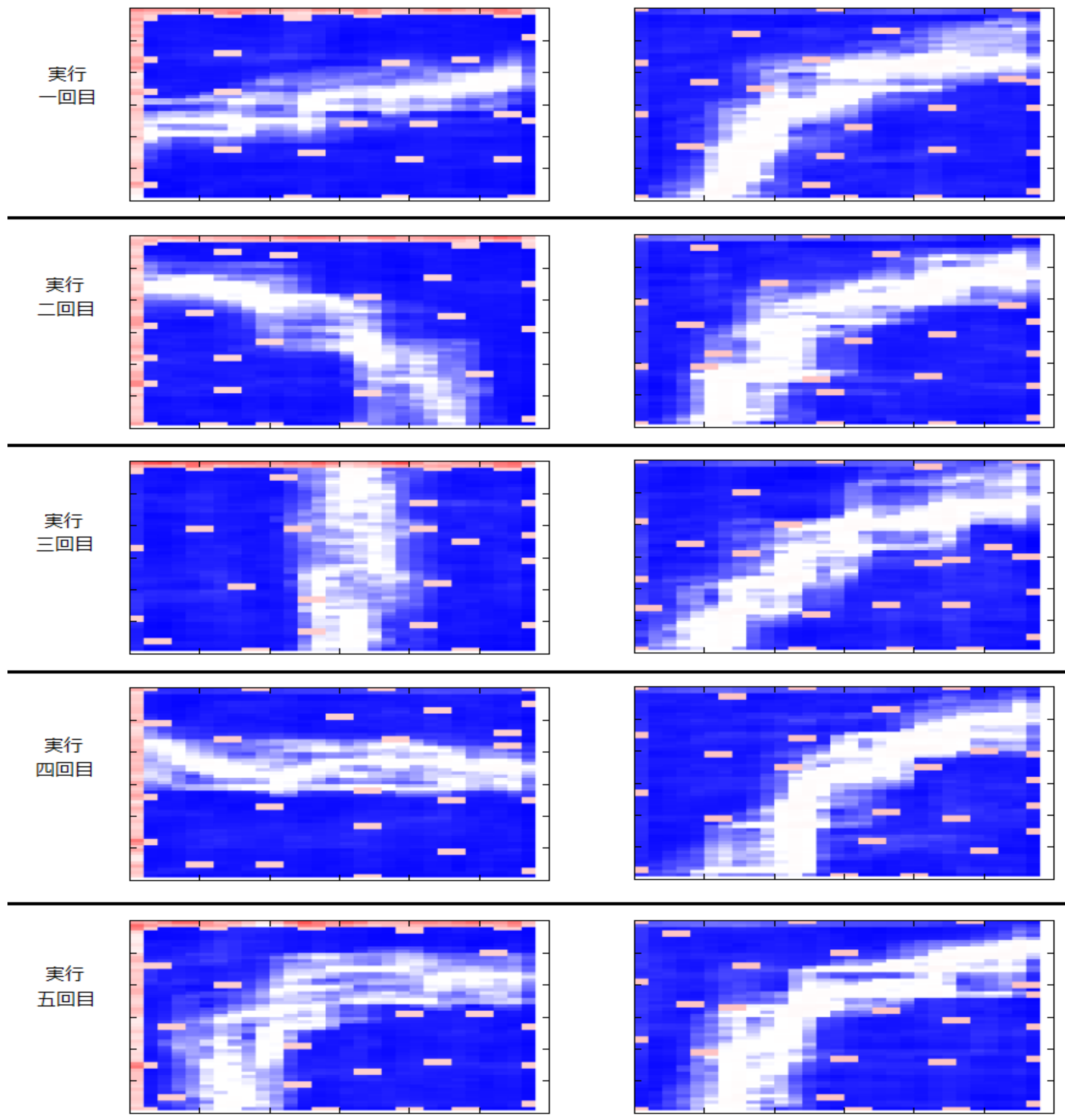


図５.既存手法と提案手法の結果

図５を見ると、既存手法では同一の学習データにもかかわらず毎回違う特徴マップが生成されており、それにより同一の入力データに対しても毎回違う位置に自己組織化されていることがわかる。一方提案手法では同一の学習データで毎回ほぼ同一の特徴マップが生成され、入力データも毎回同じような位置に自己組織化されていることがわかる。

3.3　別視点からの安定性比較実験（１）

先ほどの実験により、既存手法より提案手法の出力結果の方が安定していることが視覚的に解った。それに加え別視点からの評価として、出力結果が安定していることを定量的に示すために以下ような比較実験を行った。

先ほどの実験では入力データとして2次元×24個の二極化されたデータを用いた。今実験ではその中から出力結果の定量的な比較用のデータとして1個ずつ、計2個のデータを使用した。

以下にその2個の入力データを示す。

表１.比較用の入力データ

	1次元目の値	2次元目の値
データ1_1	2	2
データ1_2	16	16

今実験は次のようにして行った。まず、表１の二つの入力データが1000回の学習回数の内に何回抽出されたかを数える。そして抽出されるたびにその二つの入力データに対して選ばれた勝利ノードの特徴マップ上での座標と座標の選出回数も記録する。次に5回の実行回数の中で二つの入力データに対して最も勝利ノードに選出される回数の多かった特徴マップ上での座標を調べ、その座標の標準偏差を求めることで座標のばらつきを調べた。また、1000回の学習回数の内の抽出回数の中で特徴マップ上の各座標が何回勝利ノードとして抽出されたかをグラフ化した。

以下に2個の比較用入力データに対する選出回数が最も多い勝利ノード座標と標準偏差を表2、表3に、グラフを図6、図7に示す。

表2.データ1_1に対して勝利ノードに選出された回数が最も多い座標及び標準偏差

	既存手法	提案手法
実行1回目	(7, 3)	(6, 4)
実行2回目	(21, 23)	(5, 7)
実行3回目	(2, 29)	(1, 6)
実行4回目	(22, 1)	(8, 6)
実行5回目	(23, 22)	(7, 7)
標準偏差	(8.74, 11.4)	(2.42, 1.10)

※ 標準偏差の有効数字は三桁で表示している

表２から、データ1_1に対して既存手法で実行した場合の勝利ノードに選出される回数の多かった特徴マップ上での座標はx軸方向・y軸方向ともに振れ幅が大きくなっている。対して今提案手法の場合では、既存手法の場合と比較してx軸方向・y軸方向ともに振れ幅は小さい。また標準偏差が既存手法が(8.74, 11.4)、提案手法が(2.42, 1.10)と、今提案手法の方がばらつきは少ないことも分かる。

表 3.データ 1_2 に対して勝利ノードに選出された回数が最も多い座標及び標準偏差

	既存手法	提案手法
実行 1 回目	(19, 18)	(22, 15)
実行 2 回目	(10, 12)	(23, 16)
実行 3 回目	(18, 1)	(24, 14)
実行 4 回目	(20, 28)	(23, 16)
実行 5 回目	(11, 7)	(23, 15)
標準偏差	(4.22, 9.28)	(0.632, 0.748)

※ 標準偏差の有効数字は三桁で表示している

表３から、データ 1_2 に対しても同様に今提案手法の方が既存手法と比較して x 軸方向・y 軸方向ともに振れ幅は小さく、標準偏差も同様に今提案手法の方がばらつきは少ないことも分かる。

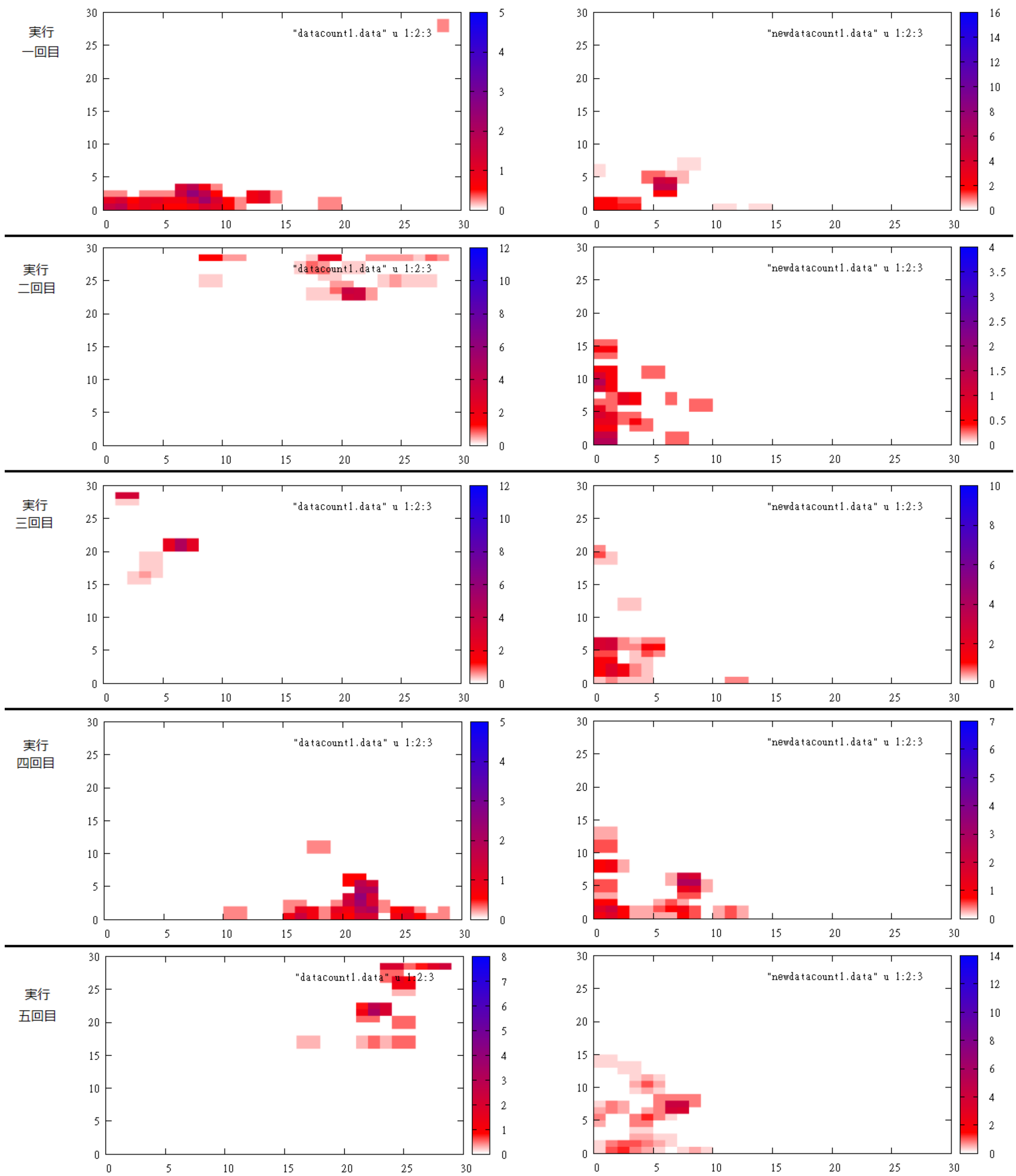


図６.データ 1_1 に対して既存手法と提案手法の勝利ノード選出数の比較

図６から、左の既存手法はデータ 1_1 に対して勝利ノードの選出回数が多い座標同士で固まってはいるものの、実行毎の位置はばらついている。対して右の提案手法はデータ 1_1 に対して勝利ノードの選出回数が多い座標同士で固まりつつ、かつ実行毎の座標位置も安定していることが分かる。

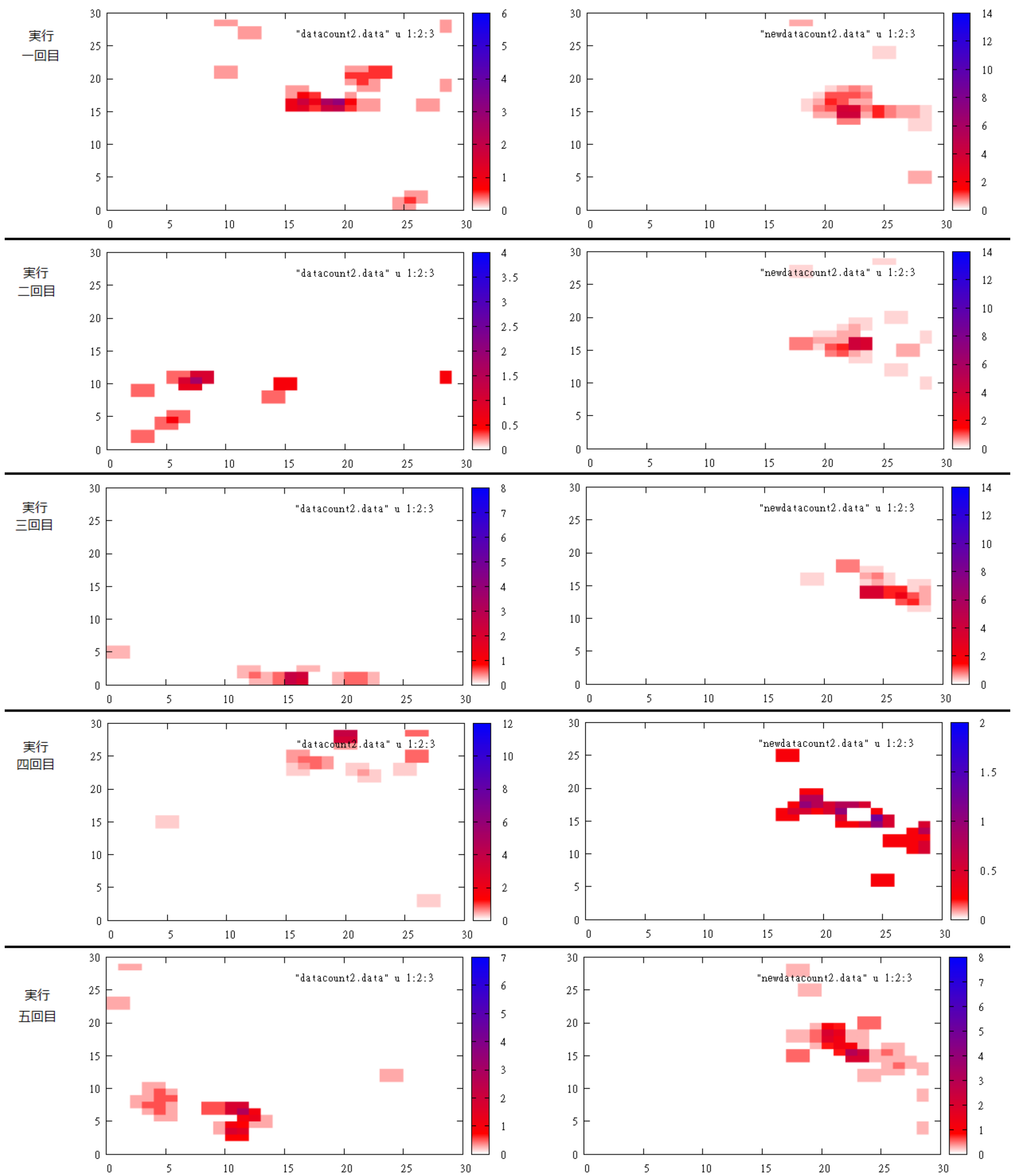


図 7 .データ 1_2 に対して既存手法と提案手法の勝利ノード選出数の比較

図 7 から、データ 1_2 に対しても同様に右の提案手法の方が左の既存手法と比較して、実行毎の勝利ノードの選出回数が多い座標の位置は安定していることが分かる。

3.4　考察（１）

3.2.2 の実験結果と 3.3 の実験結果を基に、既存手法と提案手法とで出力結果がどのように違うかを考察する。3.2.2 の実験結果を見ると肉眼でもはっきりデータのばらつき方について分かる。図５から、既存手法では実行結果毎に二極化したデータの境界線となる部分の位置が変わっていることが分かる。一方今提案手法では全ての実行結果において二極化したデータの境界線となる部分の位置がある程度同じ位置にきているように見える。

そして 3.3 の実験結果を見ると、ある入力データに注目した場合の勝利ノードの選出回数のばらつき具合が示されている。既存手法では 2 個のデータについて勝利ノード選出回数が一番多い座標の標準偏差がそれぞれ(8.74, 11.4)、(4.22, 9.28)となっている。対して今提案手法では、標準偏差がそれぞれ(2.42, 1.10)、(0.632, 0.748)となっている。このことから既存手法と比べ、今提案手法のばらつきは大幅に抑えられていることが分かる。第 2 章 2.1 より勝利ノードのばらつきは出力結果にも大きな影響を与えていることが考えられるため、結果的に今提案手法の方が出力結果の安定性に優れていることが分かる。

しかし、入力データの値の分析を抑えるという点に関しては多少疑問が残る。入力データ内での計算を行わないことで計算量が抑えられる点は満たせても、入力データに対して全ての値について参照を行っているため、入力データ数及び次元数に計算量が大きく依存する点に関しては変わりがない。

第四章　提案手法（２）

前章の 入力データ内での計算を行わないことで計算量が抑えられる点は満たせても、入力データに対して全ての値について参照を行っているため、入力データ数及び次元数に計算量が大きく依存する点に関しては変わりがないという問題点に対して、前提案手法の出力結果の安定性を損なうことがなく、入力データの分析を少なくし、計算量を抑えることを重視した手法を提案する。

4.1　入力データの参照数の決定

前章では入力データの参照数が入力データの総数に依存するという問題点が残っていたが、今提案手法では入力データの総数に依存しないような方法を考える。それには参照数について、入力データの総数に対して純粋に比例しないような参照数の決定方法が必要になる。そこで、入力データの参照によって得られる最小値・最大値に対してある程度の誤差を許容しても、勝利ノードのばらつきに影響がでないかを考えた。

まず入力データの参照方法については、無作為抽出法を採用する。無作為抽出法とは、母集団から完全に確率論的にサンプルを抽出する方法である。無作為抽出法を行う場合に必要となる標本数は以下の式で決定される。

$$\text{標本数 } n = \frac{N}{\left(\frac{E}{Z}\right)^2 \times \left(\frac{N-1}{P(1-P)}\right) + 1} \qquad (5.1)$$

(5.1)式において　N　= 母集団、　E　= 最大誤差、　Z　= 信頼度係数、　P　= 母比率である。この式によって参照数を決定する場合、入力データ数が少ない場合はその半分以上の参照数にもなるが、入力データ数が多い場合は一定値で安定する。なのでこの式を使用した場合は入力データの総数に参照数が依存しないという条件は満たすことになる。無作為抽出法は本来母集団における平均値や比率を得るために対して有力な手法であるが、最小値・最大値を取得する上でもある程度有力な手法になるのではないかと考えた。

4.2　実験方法（２）

今回の実験にあたって、第三章で使用した提案手法（１）の入力データのプログラムの参照方法の部分を(5.1)式によって得られた標本数だけ無作為抽出する今提案手法に置き換えたものを作成した。実行結果までの流れは第三章のように視覚的に出力結果の比較をするために、二極化した入力データを用いて第三章で使用した提案手法（１）のプログラムと前述の参照部分のみ今提案手法に置き換えたプログラムを複数回実行する。

また今回も第三章の 3.2 で示した図４のような二極化した入力データを用いる。なお、参照数の違いについて考察するため、第三章の 3.2 と比べ入力データの総数は多いものを使用する。

4.2.1　実験パラメータ（２）

以下のパラメータで自己組織化マップを実行する。

・実験回数　：5 回
・入力データ：3 次元×200 個のデータ、値は二極化されている
・特徴マップ：30×30 マス
・学習回数　：4000 回
・近傍領域　：20（学習回数 50 毎に 1 減少）
・学習率係数：0.01×近傍領域
・最大誤差　：0.05
・信頼度係数：1.96
・母比率　　：0.5

4.2.2　実験結果（２）

以下に二極化した入力データを用いた第三章の提案手法（１）の自己組織化マップと入力データ参照方法のみ入れ替えた今提案手法の自己組織化マップの出力結果を図８に示す。図８の左が提案手法（１）の自己組織化マップ、右が提案手法（２）の自己組織化マップである。

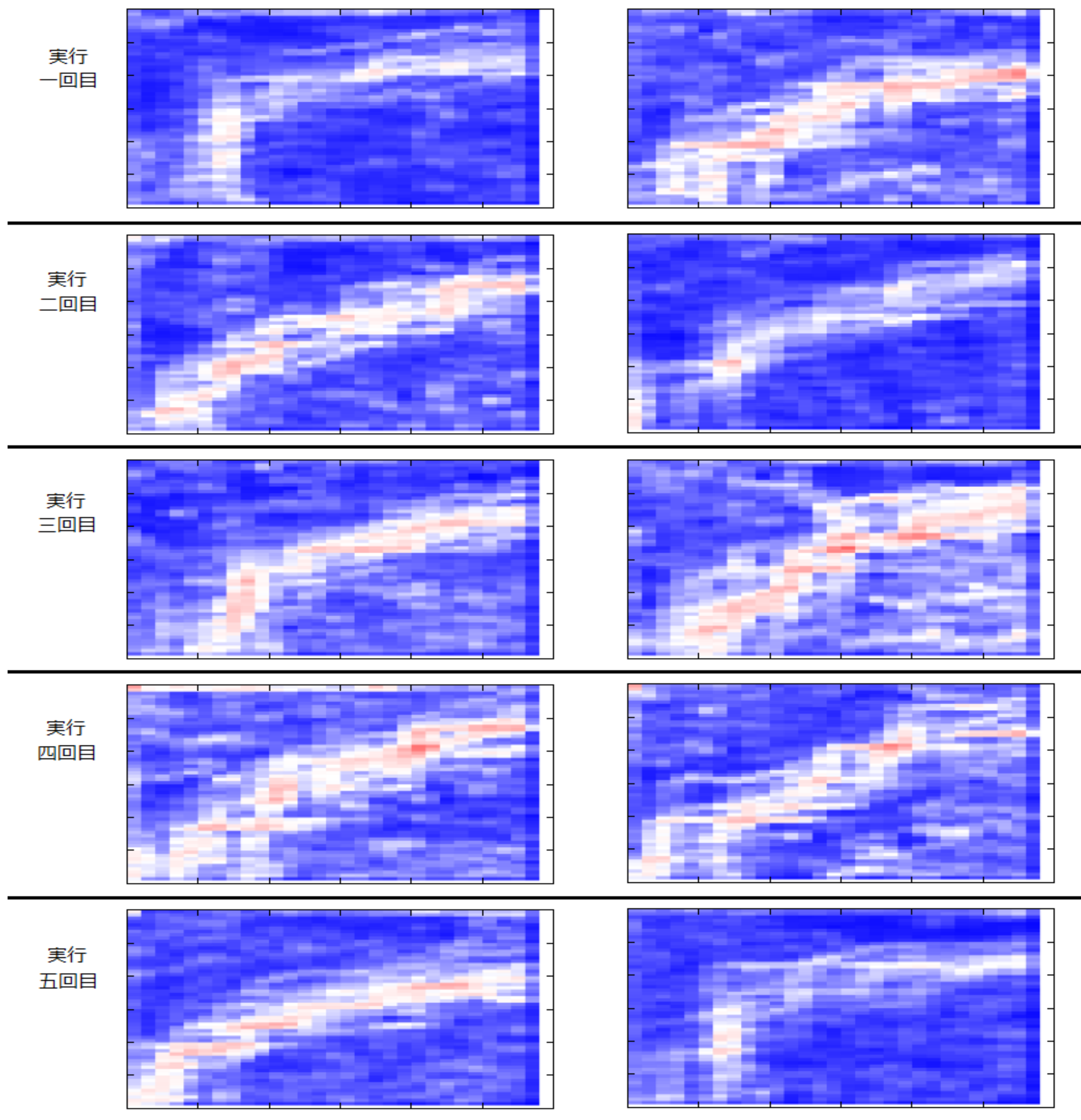


図８.提案手法（１）と提案手法（２）の結果

図８から、提案手法（１）では第三章の 3.2.2 の実験結果と同様に、二極化したデータに対して出力結果が安定していることが確認できる。一方、今提案手法である提案手法（２）でも、提案手法（１）と同様に安定していることが確認できる。

4.3　別視点からの安定性比較実験（２）

第三章の 3.3 と同様に、提案手法（１）と提案手法（２）に対して出力結果が同等に安定していることを定量的に示すために以下の比較実験を行った。

提案手法（１）と提案手法（２）の入力ベクトル参照回数を以下に示す。表４から、提案手法（２）の入力ベクトル参照回数は提案手法（１）の 39%にまで減少できていることがわかる。

表 4.各提案手法の入力ベクトル参照回数

	入力ベクトル参照回数
提案手法（１）	600
提案手法（２）	234

今実験でも第三章の 3.2.3 と同様に、入力データとして 3 次元×200 個の二極化されたデータから比較用のデータとして 1 個ずつ、計 2 個のデータを使用する。

以下にその 2 個の入力データを示す。

表 5.比較用の入力データ

	1 次元目の値	2 次元目の値	3 次元目の値
データ 2_1	4	9	2
データ 2_2	43	44	43

比較方法も第三章の 3.2.3 と同様に、実行結果の比較を以下のようにして行った。まず、表５の二つの入力データが 4000 回の学習回数の内に何回抽出されたかを数える。そして抽出されるたびにその二つの入力データに対して選ばれた勝利ノードの特徴マップ上での座標と座標の選出回数も記録する。以上を今実験と並行して行った。次に 5 回の実行回数の中で二つの入力データに対して最も勝利ノードに選出される回数の多かった特徴マップ上での座標を表化し、その座標の標準偏差を求めることで座標のばらつきを調べた。また、4000 回の学習回数の内の抽出回数の中で特徴マップ上の各座標が何回勝利ノードとして抽出されたかをグラフ化した。

以下に 2 個の比較用入力データに対する選出回数が最も多い勝利ノード座標と標準偏差を表 6、表 7 に、グラフを図 9、図 10 に示す。

表 6.データ 2_1 に対して勝利ノードに選出された回数が最も多い座標及び標準偏差

	提案手法(1)	提案手法(2)
実行 1 回目	(1, 0)	(1, 4)
実行 2 回目	(1, 0)	(3, 0)
実行 3 回目	(1, 4)	(0, 5)
実行 4 回目	(2, 9)	(1, 10)
実行 5 回目	(3, 8)	(1, 2)
標準偏差	(0.800, 3.82)	(0.980, 3.37)

※ 標準偏差の有効数字は三桁で表示している

表６から、提案手法（１）では第三章の 3.2.3 の比較結果と同様に、勝利ノードに選出される回数の多かった特徴マップ上での座標は x 軸方向・y 軸方向ともに振れ幅は小さいことがわかる。対して提案手法（２）でも、x 軸方向・y 軸方向ともに振れ幅は小さいことがわかる。また標準偏差が、提案手法（１）は(0.800, 3.82)、提案手法（２）は(0.980, 3.37)であることから、両手法を比較してばらつきに差はないということもわかる。

表 7.データ 2_2 に対して勝利ノードに選出された回数が最も多い座標及び標準偏差

	提案手法(1)	提案手法(2)
実行 1 回目	(23, 20)	(24, 22)
実行 2 回目	(23, 23)	(23, 23)
実行 3 回目	(24, 22)	(23, 25)
実行 4 回目	(24, 20)	(24, 26)
実行 5 回目	(23, 24)	(21, 22)
標準偏差	(0.490, 1.60)	(1.10, 1.62)

※ 標準偏差の有効数字は三桁で表示している

表７でも同様に、提案手法（１）と提案手法（２）共に、x 軸方向・y 軸方向ともに振れ幅は小さく、標準偏差も提案手法（１）と提案手法（２）ではばらつきに差はないということもわかる。

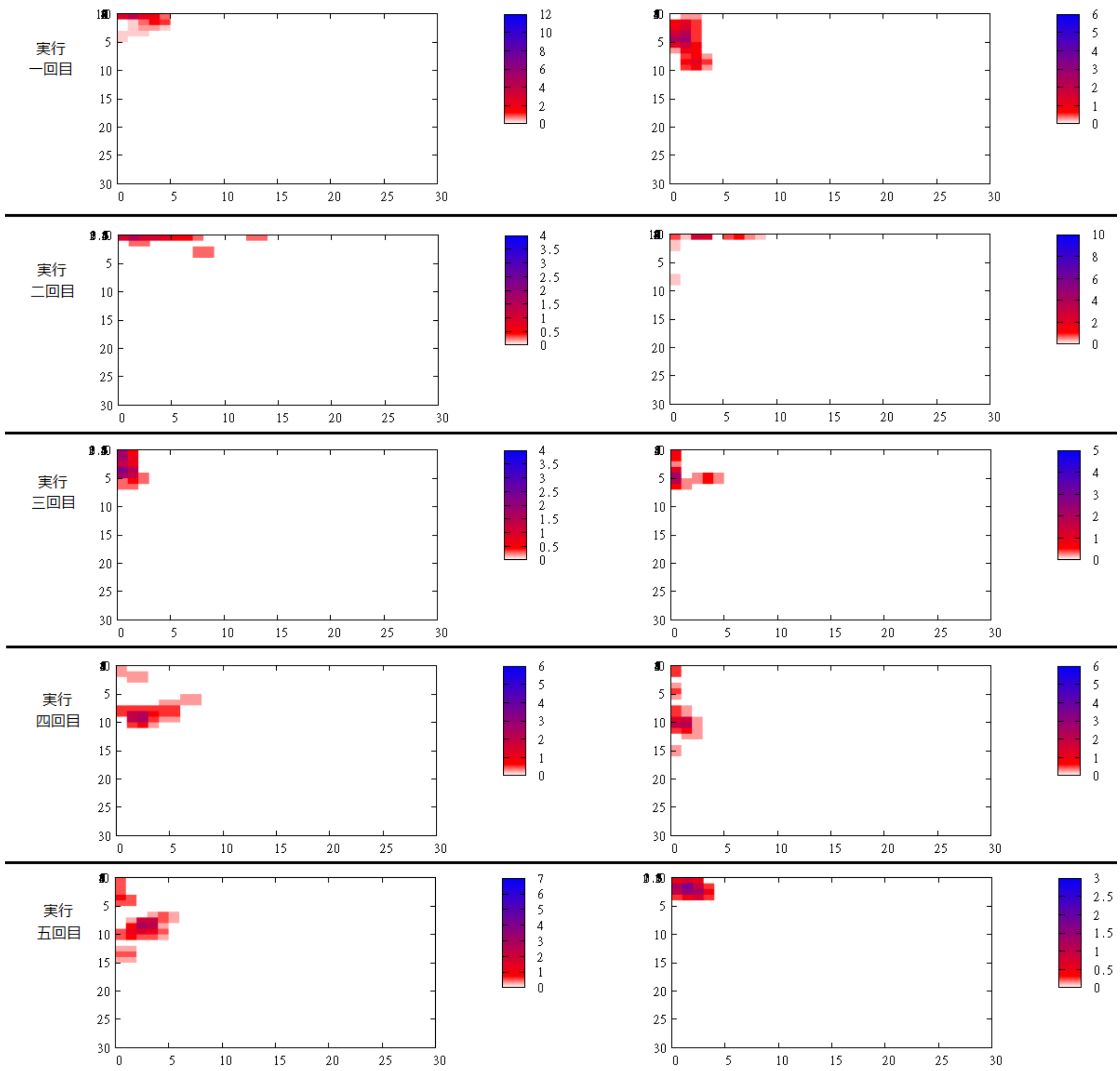


図９.データ 2_1 に対して既存手法と提案手法の勝利ノード選出数の比較

図９から、左の提案手法（１）では第三章の 3.2.3 の比較結果と同様に、データ 2_1 に対して勝利ノードの選出回数が多い座標同士で固まっており、かつ実行毎の座標位置も安定していることが分かる。対して右の提案手法（２）でも提案手法（１）と同様に勝利ノードの座標位置が安定していることが分かる。

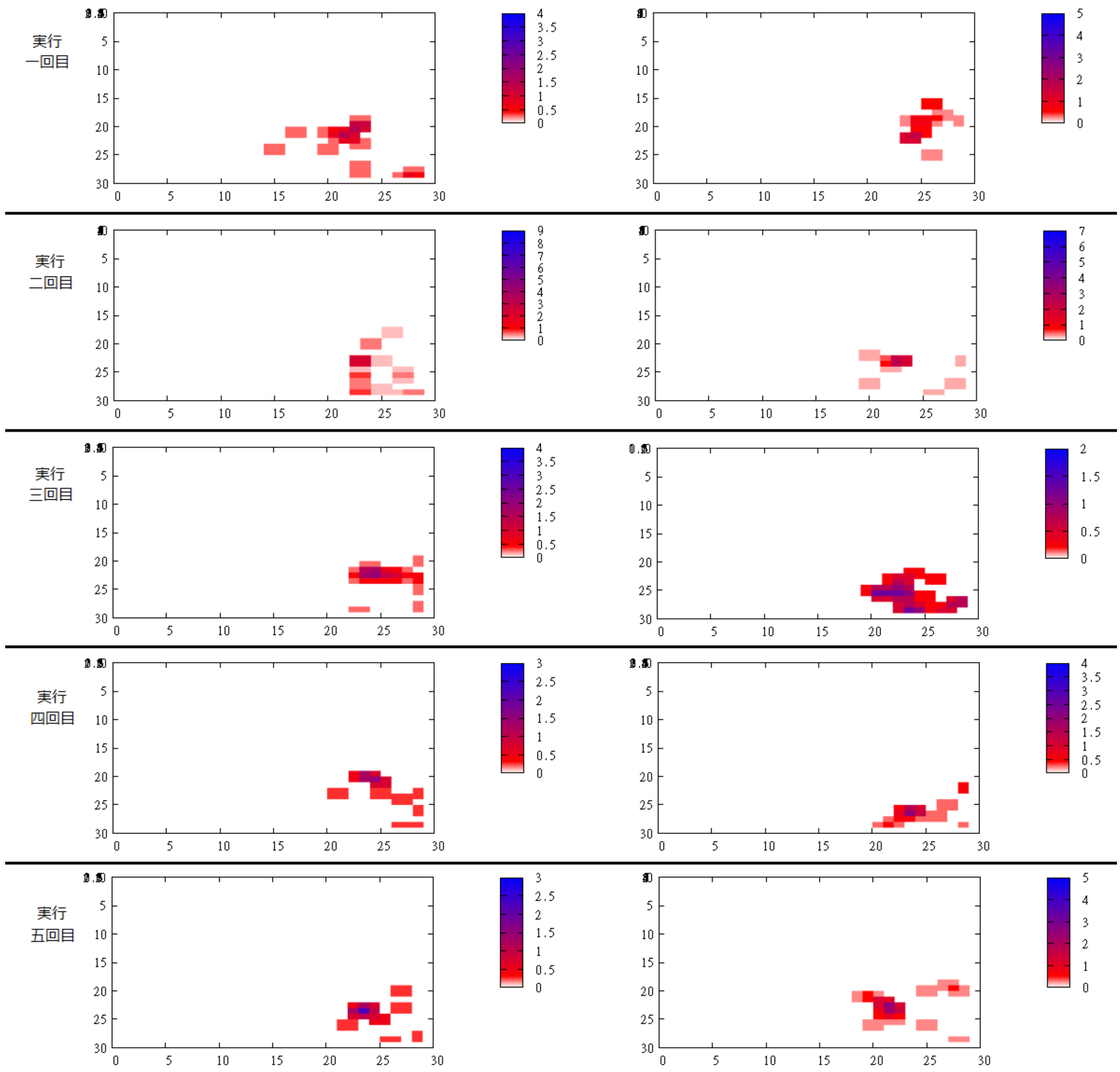


図１０.データ 2_2 に対して既存手法と提案手法の勝利ノード選出数の比較

図１０でも同様に、左の提案手法（1）と右の提案手法（2）は両方とも勝利ノードの選出回数が多い座標同士で固まっており、かつ勝利ノードの座標位置も安定していることが分かる。

4.4　考察（２）

4.2.2 の実験結果と 4.3 の実験結果を基に、提案手法（１）と提案手法（２）とで出力結果がどのように違うかを考察する。まず 4.2.2 の実験結果について、提案手法（１）はデータ数の関係から出力結果が荒くなっているが、境界線は同じような位置にあることを視覚的に判断できる程度となっている。提案手法（２）についても提案手法（１）と同様に境界線は同じような位置にあることを視覚的に判断できる。

4.3 の実験結果について、提案手法（２）の入力ベクトル参照数は提案手法（１）と比べて半分以下に抑えることができている。勝利ノードの選出回数が最も多いものの比較については、標準偏差を見ると、提案手法（１）がそれぞれ(0.800, 3.82)、(0.490, 1.60)となっていて、提案手法（２）がそれぞれ(0.980, 3.37)、(1.10, 1.62)となっている。第三章の既存手法の標準偏差と比較すると両提案手法とも十分にばらつきが少ないと考えられる。したがって、提案手法（２）に関して入力ベクトルの参照数に対して入力ベクトルの総数への依存を抑えつつ、提案手法（１）と同等の安定性を保つという目的は達成できたと考えられる。

第五章　おわりに

本稿では既存手法の乱数による初期化の出力結果へのばらつきの影響を改善するための初期化範囲指定と特徴マップのソートを導入した手法を提案した。また入力データの分析を抑えるためと参照数の入力データの総数への依存を抑えるために、無作為抽出法を用いた手法を提案した。

考察より、既存手法と比較して出力結果が安定して得られることが考えられる。また、入力データの参照数を少なくした場合でもその前と同等の安定性を得られることが考えられる。しかし二極化以外の入力データに対しては勝利ノードの位置が安定していることを示せたとしても、視覚的にそれを十分判断できるかどうかは難しい。

今後の展望として、境界線が表しづらい出力結果に対しても何らかの基準からクラスタリングを行えるような手法を考え、本稿で提案した手法と組み合わせることで、より安定的な出力結果を得られるような研究を進めていく。

謝辞

本研究を進めるにあたり、様々なご指導を頂きました三好力教授に深謝いたします。
また発表を通じて多くの示唆を頂いた三好研究室の皆様に感謝します。

参考文献

1) 徳高平蔵, 大北正昭, 藤村喜久郎
自己組織化マップとその応用（シュプリンガー・ジャパン株式会社, 2007）

2) 大北正昭, 徳高平蔵, 藤村喜久郎, 権田英功
自己組織化マップとツール（シュプリンガー・ジャパン株式会社, 2008）

3) Mu-Chun Su, Hsiao-Te Chang,
Fast Self-Organizing Feature Map Algorithm
(IEEE Transactions on Neural Networks, 11(3):721--733 , MAY 2000)

4) 三好力
学習データによる初期ノード交換を用いた SOM の特徴マップ初期化
(日本知能情報ファジィ学会誌 Vol.9 No.2 pp.000-000, 2007)

5) 世論調査におけるサンプリング数の決定
http://www.wound-treatment.jp/next/wound225.htm

6) Toolbox:Sampling
http://www.mahoroba.ne.jp/~felix/Toolbox/Methods/Statistics/sampling.html

7) 標準偏差とは
http://www.hinkai.com/qc/hensa.html

8) GNUPLOT
http://t16web.lanl.gov/Kawano/gnuplot/index.html

付録

ソースコード(1)

```
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.StreamTokenizer;
import java.text.DecimalFormat;
import java.util.Random;

public class SomOutNew {
static DecimalFormat df = new DecimalFormat();
static int mapsize = 30; //出力マップの範囲
static double[][] w_average = new double[mapsize][mapsize]; //平均値格納用変数
static int sort_nx;
static double[][][] w = new double[mapsize][mapsize][1000]; //参照ベクトル格納用変数

public static void main(String args[]){
Random rand = new Random(); //ランダム生成用
double[][] u = new double[mapsize][mapsize]; //ユークリッド距離格納用変数
int winx=0, winy=0; //勝利ノードの座標格納用変数
double a = 0.01; //学習率係数
int nearcount = 0; //学習回数毎に近傍領域を狭めていくための変数
int near = 20; //近傍領域
int nt; //入力ベクトルランダム抽出用変数
int learncount=4000; //学習回数
double[][] ux = new double[mapsize][mapsize]; //学習完了時、左右の参照ベクトルのユークリッド距離格納用変数
double[][] uy = new double[mapsize][mapsize]; //学習完了時、上下の参照ベクトルのユークリッド距離格納用変数
int inputsize = 200; //入力ベクトルの個数
int sampling_min = 0; //特定の小さい入力ベクトルが何回抽出されたか数える変数
int[][] wincount_min = new int[mapsize][mapsize]; //勝利ノード座標が何回選ばれたか数える変数
int sampling_max = 0; //特定の大きい入力ベクトルが何回抽出されたか数える変数
int[][] wincount_max = new int[mapsize][mapsize]; //勝利ノード座標が何回選ばれたか数える変数

for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
wincount_min[i][j]=0;
wincount_max[i][j]=0;}}

/*入力ベクトル読み込み*/
System.out.println("入力ベクトル");
double m[][]=new double[5000][100];
int nx=0, ny=0;
try{
FileReader fr=new FileReader("C:/inputMore-b.data"); //FileReader オブジェクトの作成
StreamTokenizer st=new StreamTokenizer(fr); //StreamTokenizer オブジェクトの作成
st.eolIsSignificant(true);
while(st.nextToken()!=StreamTokenizer.TT_EOF){ //ファイルの中身の終わりまでループ
switch(st.ttype){ //読み取った文字列に対して、
        case StreamTokenizer.TT_NUMBER: //数字だった場合
        m[ny][nx]=st.nval; //読み取ったデータを配列に代入
        nx++;
        break;

        case StreamTokenizer.TT_EOL: //改行だった場合
        ny++;
        nx=0;
        break;
}}
fr.close();}
catch(Exception e){System.out.println(e);}

/*学習データ全参照パターン*/
int large_x=0, large_y=0; //学習データ最大値の座標格納用変数
int small_x=0, small_y=0; //学習データ最小値の座標格納用変数
for(int i=0 ; i<ny ; i++){
for(int j=0 ; j<nx ; j++){
        if(m[large_x][large_y] < m[i][j]){
        large_x = i; large_y = j;}
        if(m[small_x][small_y] > m[i][j]){
        small_x = i; small_y = j;}
}}

/*特徴マップ初期化*/
double r_max=m[large_x][large_y], r_min=m[small_x][small_y]; //学習データ最大,最小値格納用変数
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
for(int k=0 ; k<nx ; k++){
        w[i][j][k] = Math.floor(Math.random()*(r_max-r_min+1))
+r_min; //乱数代入(範囲指定)
}}}

/*特徴マップソート*/
double w_sum=0; //次元の値を合計したものを一時格納するための変数
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
for(int k=0 ; k<nx ; k++){
        w_sum = w_sum+w[i][j][k];
}
        w_average[i][j] = w_sum/nx;
        w_sum=0;
}}

for(int g=0;g<mapsize;g++) quick_first(0, mapsize-1, g); //縦
for(int g=0;g<mapsize;g++) quick_second(0, mapsize-1, g); //横
/*----特徴マップ初期化完了----*/

/**学習開始**/
while(learncount>0){
nt = rand.nextInt(ny+1); //抽出対象を乱数で決定

/*ユークリッド距離計算*/
double sum=0;
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
for(int k=0; k<nx; k++){
        sum = sum + Math.pow((w[i][j][k] - m[nt][k]),2);
}
        u[i][j] = Math.sqrt(sum);
        sum=0;
}}

/*勝利ノード判定*/
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
if(u[winx][winy]>u[i][j]){winx = i;winy = j;}
}}

/*比較用に、特定の入力ベクトルの勝利ノード座標を格納*/
if(nt==5){sampling_min++; wincount_min[winx][winy]++;}
if(nt==195){sampling_max++; wincount_max[winx][winy]++;}

/*参照ベクトル更新*/
for(int k=0; k<nx; k++){
        w[winx][winy][k] = w[winx][winy][k] + a*near*(m[nt][k]-
w[winx][winy][k]);
}
for(int i=1 ; i<=near ; i++){
        if(mapsize>winx+i){
        for(int k=0; k<nx; k++){
```

```
sort_nx=nx;
			w[winx+i][winy][k] = w[winx+i][winy][k] +
a*near*(m[nt][k]-w[winx+i][winy][k]);}
		}
		if(0<winx-i){
		for(int k=0; k<nx; k++){
			w[winx-i][winy][k] = w[winx-i][winy][k] +
a*near*(m[nt][k]-w[winx-i][winy][k]);}
		}
		if(mapsize>winy+i){
		for(int k=0; k<nx; k++){
			w[winx][winy+i][k] = w[winx][winy+i][k] +
a*near*(m[nt][k]-w[winx][winy+i][k]);}
		}
		if(0<winy-i){
		for(int k=0; k<nx; k++){
			w[winx][winy-i][k] = w[winx][winy-i][k] +
a*near*(m[nt][k]-w[winx][winy-i][k]);}
		}
}
if(near>=2){
for(int i=1 ; i<=near-1 ; i++){
for(int j=1 ; j<=near-1 ; j++){
		if(mapsize>winx+i && mapsize>winy+j){
		for(int k=0; k<nx; k++){
			w[winx+i][winy+j][k] = w[winx+i][winy+j][k] +
a*near*(m[nt][k]-w[winx+i][winy+j][k]);}
		}
		if(mapsize>winx+i && 0<winy-j){
		for(int k=0; k<nx; k++){
			w[winx+i][winy-j][k] = w[winx+i][winy-j][k] +
a*near*(m[nt][k]-w[winx+i][winy-j][k]);}
		}
		if(0<winx-i && mapsize>winy+j){
		for(int k=0; k<nx; k++){
			w[winx-i][winy+j][k] = w[winx-i][winy+j][k] +
a*near*(m[nt][k]-w[winx-i][winy+j][k]);}
		}
		if(0<winx-i && 0<winy-j){
		for(int k=0; k<nx; k++){
			w[winx-i][winy-j][k] = w[winx-i][winy-j][k] +
a*near*(m[nt][k]-w[winx-i][winy-j][k]);}
		}
}}}

learncount--; nearcount++;
if(nearcount%50==0) near--; //近傍領域減少
}
/**----学習終了----**/


/*各ベクトル間ユークリッド距離計算(横)*/
double sumx = 0;
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize-1 ; j++){
for(int k=0; k<nx; k++){
		sumx = sumx + Math.pow((w[i][j][k] - w[i][j+1][k]),2);
}
		ux[i][j] = Math.sqrt(sumx);
		sumx=0;
}}
/*追加データ x*/
for(int i=0 ; i<mapsize ; i++){ux[i][mapsize-1]=0;}

/*各ベクトル間ユークリッド距離計算(縦)*/
double sumy = 0;
for(int i=0 ; i<mapsize-1 ; i++){
for(int j=0 ; j<mapsize-1 ; j++){
for(int k=0; k<nx; k++){
		sumy = sumy + Math.pow((w[i][j][k] - w[i+1][j][k]),2);
}
		uy[i][j] = Math.sqrt(sumy);
		sumy=0;
}}
/*追加データ y*/
for(int i=0 ; i<mapsize ; i++){uy[mapsize-1][i]=0;}

/*上下と左右の学習ベクトルのユークリッド距離を合成*/
int ix=0,iy=0,map2;
map2=mapsize*2;
double[][] uxy = new double[map2][map2]; //合成用配列
for(int i=0 ; i<map2 ; i++){
for(int j=0 ; j<map2 ; j++){
		if(i%2 == 0){
		if(j<mapsize){
			uxy[i][j] = ux[ix][j];
		}else{
			uxy[i][j] = 0; //余りを 0 で補完
		}}
		else if(i%2 == 1){
		if(j<mapsize){
			uxy[i][j] = uy[iy][j];
		}else{
			uxy[i][j] = 0; //余りを 0 で補完
		}}
}
		if(i%2 == 0){ix++;}
		else if(i%2 == 1){iy++;}
}

/*入力ベクトルを参照ベクトル上に組み込む*/
double[][][] u_input = new double[mapsize][mapsize][inputsize]; //
ユークリッド距離格納用変数
double sum_input=0;
int piece_input=0;

/*入力ベクトル一つひとつの参照ベクトルとのユークリッド距離
を全て求める*/
while(piece_input<inputsize){
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
for(int k=0; k<nx; k++){
		sum_input = sum_input + Math.pow((w[i][j][k] -
m[piece_input][k]),2);
}
		u_input[i][j][piece_input] = Math.sqrt(sum_input);
		sum_input=0;
}}
		piece_input++;
}

piece_input=0;
winx=0;
winy=0;
/*各勝利ノードを求める*/
while(piece_input<inputsize){
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
		if(u_input[winx][winy][piece_input]>u_input[i][j]
[piece_input]){winx = i; winy = j;}
}}

/*入力ベクトルをマーク*/
if(winx==0 && winy!=0){//上がない場合
		if(uxy[(winx*2)][winy-1]<=uxy[(winx*2)][winy] &&
uxy[(winx*2)][winy-1]<=uxy[(winx*2)+1][winy]){
			uxy[(winx*2)][winy-1]=100;
		}else if(uxy[(winx*2)][winy]<uxy[(winx*2)][winy-1] &&
uxy[(winx*2)][winy]<=uxy[(winx*2)+1][winy]){
			uxy[(winx*2)][winy]=100;
		}else if(uxy[(winx*2)+1][winy]<uxy[(winx*2)][winy-1]
&& uxy[(winx*2)+1][winy]<uxy[(winx*2)][winy]){
			uxy[(winx*2)+1][winy]=100;
		}
}else if(winy==0 && winx!=0){//左がない場合
		if(uxy[(winx*2)-1][winy]<=uxy[(winx*2)][winy] &&
uxy[(winx*2)-1][winy]<=uxy[(winx*2)+1][winy]){
			uxy[(winx*2)-1][winy]=100;
		}else if(uxy[(winx*2)][winy]<uxy[(winx*2)-1][winy] &&
```

```
uxy[(winx*2)][winy]<=uxy[(winx*2)+1][winy]){
				uxy[(winx*2)][winy]=100;
		}else if(uxy[(winx*2)+1][winy]<uxy[(winx*2)-1][winy]
&& uxy[(winx*2)+1][winy]<uxy[(winx*2)][winy]){
				uxy[(winx*2)+1][winy]=100;
		}
}else if(winx==0 && winy==0){//座標が 0,0 の場合
		if(uxy[(winx*2)][winy]<=uxy[(winx*2)+1][winy]){
				uxy[(winx*2)][winy]=100;
		}else if(uxy[(winx*2)+1][winy]<uxy[(winx*2)][winy]){
				uxy[(winx*2)+1][winy]=100;
		}
}else{
		if(uxy[(winx*2)-1][winy]<=uxy[(winx*2)][winy-1] &&
uxy[(winx*2)-1][winy]<=uxy[(winx*2)][winy] && uxy[(winx*2)-1]
[winy]<=uxy[(winx*2)+1][winy]){
				uxy[(winx*2)-1][winy]=100;
		}else if(uxy[(winx*2)][winy-1]<uxy[(winx*2)-1][winy] &&
uxy[(winx*2)][winy-1]<=uxy[(winx*2)][winy] && uxy[(winx*2)]
[winy-1]<=uxy[(winx*2)+1][winy]){
				uxy[(winx*2)][winy-1]=100;
		}else if(uxy[(winx*2)][winy]<uxy[(winx*2)-1][winy] &&
uxy[(winx*2)][winy]<uxy[(winx*2)][winy-1] && uxy[(winx*2)]
[winy]<=uxy[(winx*2)+1][winy]){
				uxy[(winx*2)][winy]=100;
		}else if(uxy[(winx*2)+1][winy]<uxy[(winx*2)-1][winy]
&& uxy[(winx*2)+1][winy]<uxy[(winx*2)][winy-1] &&
uxy[(winx*2)+1][winy]<uxy[(winx*2)][winy]){
				uxy[(winx*2)+1][winy]=100;
		}
}
winx=0;
winy=0;
piece_input++;
}


/*学習結果を data ファイルで出力*/
try{
BufferedWriter bw = new BufferedWriter(new FileWriter("c:\\som-
data-file\\outputMore-b.data")); //FileWriter オブジェクトの作成

for(int i=0 ; i<map2 ; i++){
for(int j=0 ; j<mapsize ; j++){
		Double dNumberx = new Double(uxy[i][j]); //write に
double 型を対応させる
		bw.write(j+" "+i+" "+dNumberx.toString()); //ファイルへ
の書き込み
		bw.newLine();
}bw.newLine();
}
bw.close(); //ファイルを閉じる
}
catch(Exception e){System.out.println(e); }


/*入力ベクトル呼び出し回数を data ファイルで出力*/
try{
BufferedWriter bw = new BufferedWriter(new FileWriter("c:\\som-
data-file\\newdatacount1.data")); //FileWriter オブジェクトの作成

for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
		bw.write(i+" "+j+" "+wincount_min[i][j]); //ファイルへの
書き込み
		bw.newLine();
}bw.newLine();
}
bw.close(); //ファイルを閉じる
}
catch(Exception e){System.out.println(e); }
try{
BufferedWriter bw = new BufferedWriter(new FileWriter("c:\\som-
data-file\\newdatacount2.data")); //FileWriter オブジェクトの作成
for(int i=0 ; i<mapsize ; i++){
for(int j=0 ; j<mapsize ; j++){
		bw.write(i+" "+j+" "+wincount_max[i][j]); //ファイルへの
書き込み
		bw.newLine();
}bw.newLine();
}
bw.close(); //ファイルを閉じる
}
catch(Exception e){System.out.println(e); }


/**----SOM データ出力完了----**/
}


/**縦方向クイックソート用メソッド**/
static void quick_first(int m, int n, int g){
int f_loop=m, b_loop=n;
double t;
double[] wt = new double[sort_nx];
int half=(m+n)/2;
double ki=w_average[(int)half][g];
while(f_loop <= b_loop){
		while(w_average[f_loop][g]<ki) f_loop++;
		while(w_average[b_loop][g]>ki) b_loop--;
		if(f_loop <= b_loop){
				t=w_average[f_loop][g]; w_average[f_loop]
[g]=w_average[b_loop][g]; w_average[b_loop][g]=t;
				for(int h=0;h<sort_nx;h++){
						wt[h]=w[f_loop][g][h]; w[f_loop][g]
[h]=w[b_loop][g][h]; w[b_loop][g][h]=wt[h];
				}f_loop++; b_loop--;
				}
		}
		if(m<b_loop) quick_first(m, b_loop, g);
		if(f_loop<n) quick_first(f_loop, n, g);
}


/**横方向クイックソート用メソッド**/
static void quick_second(int m, int n, int g){
int f_loop=m, b_loop=n;
double t;
double[] wt = new double[sort_nx];
int half=(m+n)/2;
double ki=w_average[g][(int)half];
while(f_loop <= b_loop){
		while(w_average[g][f_loop]<ki) f_loop++;
		while(w_average[g][b_loop]>ki) b_loop--;
		if(f_loop <= b_loop){
				t=w_average[g][f_loop]; w_average[g]
[f_loop]=w_average[g][b_loop]; w_average[g][b_loop]=t;
				for(int h=0;h<sort_nx;h++){
						wt[h]=w[g][f_loop][h]; w[g][f_loop]
[h]=w[g][b_loop][h]; w[g][b_loop][h]=wt[h];
				}f_loop++; b_loop--;
				}
		}
		if(m<b_loop) quick_second(m, b_loop, g);
		if(f_loop<n) quick_second(f_loop, n, g);
}
}
```

ソースコード**(2)**
※ 学習データ全参照パターンを以下のコードに置き換えた

```
/*学習データ無作為抽出パターン*/
int large_x=0, large_y=0; //入力データ最大値の座標格納用変数
int small_x=0, small_y=0; //入力データ最小値の座標格納用変数
int check[][] = new int[ny+1][nx]; //重複チェック用変数
for(int i=0 ; i<ny+1 ; i++){
for(int j=0 ; j<nx ; j++){
		check[i][j] = 0; //重複しているかどうかを0,1で判断するため、初期化
}}

/*無作為抽出における標本数の決定*/
int N = (ny+1)*nx; //母集団の数
double E = 0.05; //最大誤差
double Z = 1.96; //信頼係数0.95を基にした値
double P = 0.5; //予想される母平均の比率
int n; //必要な標本数
n=(int)(N/((Math.pow(E/Z,2)*((N-1)/(P*(1-P))))+1));

/*標本数分入力データの次元から値を抽出・比較し、最大最小値を取得*/
int R_input_ny, R_input_nx;
int overlap=0;
for(int i=0 ; i<n ; i++){
		overlap++;
		R_input_ny = (int) Math.floor(Math.random()*(ny+1));
		R_input_nx = (int) Math.floor(Math.random()*(nx));

		switch(check[R_input_ny][R_input_nx]){
		case 0: //重複してない場合
		if(m[large_x][large_y] < m[R_input_ny][R_input_nx]){
				large_x = R_input_ny;
				large_y = R_input_nx;
		}
		if(m[small_x][small_y] > m[R_input_ny][R_input_nx]){
				small_x = R_input_ny;
				small_y = R_input_nx;
		}
		check[R_input_ny][R_input_nx] = 1;
		break;

		case 1: //重複してる場合
		i--;
		break;
		}
}
```