TiliNG

A new Growing Neural Gas variant for density insensitive state space generation and its applications

Attila Kádár, Márk Dániel Szalai

Supervisor:
Gábor Horváth
External supervisor:
Péter Kovács

Budapest University of Technology and Economics
Department of Networked Systems and Services
Budapest, Hungary
October 2020

Contents

1 Introduction **3**

2 Related works **4**
- 2.1 Basic GNG algorithm . . . 4
- 2.2 GNG with utility . . . 5
- 2.3 Adaptive Incremental Neural Gas . . . 6
- 2.4 Summary . . . 7

3 TiliNG **10**
- 3.1 Steps of the algorithm . . . 10
  - 3.1.1 Train . . . 11
  - 3.1.2 Adaptation . . . 11
  - 3.1.3 Deleting . . . 11
  - 3.1.4 The role of the sample-counter . . . 12
- 3.2 Approximate solution of adaptation . . . 13
  - 3.2.1 Fix-point iteration with KKT . . . 13
  - 3.2.2 Fix-point iteration without constraints . . . 15
- 3.3 Graph structure . . . 17
- 3.4 Efficiency tricks, Implementation . . . 19
  - 3.4.1 Spatial search . . . 20
  - 3.4.2 Pseudo-code . . . 20
- 3.5 Scalability and complexity . . . 21

4 Proposed approximation of parameter R **22**
- 4.1 Trial and error . . . 22
- 4.2 Multi-normal distribution . . . 23
- 4.3 Combined method . . . 25

5 Evaluation and results **26**
- 5.1 Evaluation of TiliNG . . . 26
  - 5.1.1 2D data sets . . . 26
  - 5.1.2 3D data set . . . 28
- 5.2 Comparison with GNG variants . . . 29
  - 5.2.1 Uniform density . . . 29
  - 5.2.2 Remembering past behaviours of a system . . . 30

6 Applications **33**
- 6.1 Clustering . . . 33
  - 6.1.1 TiliNG as a useful tool for clustering algorithms . . . 33
  - 6.1.2 Comparison with well-know clustering algorithms . . . 34
- 6.2 Anomaly detection . . . 37
- 6.3 Wireless Sensor Network Topology and TiliNG . . . 39

1 Introduction

Imagine having a huge amount of samples (data points in $\mathbb{R}^n$) and having the task of analyzing them. As the number of samples keeps growing it is getting more and more important to have a compact representation of the samples, which enables faster analysis and makes it easier to keep the relevant information available. Ideally such a compact representation is capable of following data trends, is fine enough and keeps as much information as possible about the samples.

Growing Neural Gas (GNG) algorithms have been known and used for decades, since they offer a simple and fast solution to represent and cluster huge number of observations with small number of discrete states in real time. They work by processing the samples one by one, while building a graph- or cloud-like structure with nodes (that represents a subset of the samples) and edges (that represents a connection, i.e. closeness between subsets of samples). These nodes and edges can be deleted and new ones can be constructed, in order to make the graph structure more adaptive, resulting in a better sample representation.

The original GNG algorithm and the variants published so far have the following deficiencies:

- In case of certain applications the distribution of the observations is not stationary. The behaviour of the system can change, evolve in time dynamically. In general most of the GNG algorithms can only adapt to these changes (evolutions) by forgetting the past behaviour of the system.
- Most of the GNG variants use many hyper-parameter, which can be adjusted only by heuristic methods. The final state representation of the observations highly depends on these parameters.
- The existing GNG variants cover the dense parts of the observation space with more states, while rare observations can potentially be ignored completely. This behavior can be a disadvantage in case of several applications.

We propose a GNG based algorithm, that can cope with the above mentioned problems, therefore resulting in a better state-space representation. We pay special attention to the handling of non stationary behaviour in such a way, that the previously built state-space does not suffer any consequences. We also aim for a state-space with a density, that is independent of the samples' density. This way the rarely occurring observations are actually represented at the end of the algorithm and are not lost between the more frequent states. Finally, most of the GNG variants use a bunch of hyper-parameters, we would like to minimise their number. This does not only eliminate the uncertainty they introduce into the system, but also results in an easier and more comfortable to use tool. With these properties we hope to open our GNG variant to the effective application on fields such as high speed clustering and anomaly detection.

2 Related works

2.1 Basic GNG algorithm

The basic GNG has been originally introduced in 1995 in [1]. It is able to learn the important topological relations in a given set of input vectors (samples) by means of a simple Hebb-like learning rule.

Below we summarize the main steps of the algorithm:

0. Start with two nodes a and b at random positions W_a and W_b in R^n.

When a new sample is given to the algorithm, the following steps are taken:

1. The closest and the second closest node is selected (denoted by s and t)
2. The error measure associated to the nodes storing the accumulated errors are updated for s, it is incremented by the distance between the new sample and s.
3. The position of node s and its neighbours are updated, they are moved towards the new sample. The amount of movement (the learning rate) is usually set to $\varepsilon_b = 0.05$ for the closest node and $\varepsilon_n = 0.0006$ for the neighbours. No other nodes are moved.
4. The edges are maintained. Nodes s and t are connected with a new edge, or if there is already such an edge in the graph, its age is re-set to 0. The age of the edges between s and its neighbours are incremented.
5. Then the edge removal step follows. Edges with age greater than a predefined maximum value are removed (typical threshold: $a_{max} = 80$). Nodes that became isolated are removed.
6. If the current iteration is the integer multiple of λ (typically $\lambda = 300$), then a new node is created. The new node will be half-way between the node having the largest error and its neighbour having the largest error. The error of the involved nodes is then reduced by the factor of $\alpha = 0.05$.
7. The errors of the nodes are multiplied by 1 - β, with $\beta = 0.0005$

This GNG algorithm suffers from several problems:

- Only the age of Best Matching Units' (BMU's) neighboring edges are increased. In case of sudden change of the sample distribution many nodes can become dead, their edge age is not incremented, hence they will never be removed and will be useless.
- This algorithm uses many hyper-parameters, for example a few of the above mentioned parameters are: ε_b, ε_n, a_{max}, λ, etc. In perspective of efficiency it is necessary to use different parameters for different observation spaces and there is no clear method about how to set these parameters optimally. (Most of the people use the algorithm with the default

hyper-parameters, as described in [1], regardless of their goals, which is problematic.)

Consequently, the basic GNG can not be used for non-stationary sample distributions.

2.2 GNG with utility

To treat the dead node problem of the basic GNG, the GNG algorithm was modified by assigning utility to the nodes. This extension, called GNG-U, appeared in [2]. What GNG-U does is that it removes nodes that contribute little to the reduction of error, in favour of inserting them where they would contribute more to the reduction of the error. To do that, instead of accumulating the errors in the nodes, we associate a new quantity, the utility U_s, for every node. Utility of s: the increase in the mean squared error for the sample if node s was non-existent. An approximation of how much a particular node s decreases the error of the input signals in its region. Removing a high-utility node, compared to a low-utility node, means the increase in error in that region will be greater.

Most steps of the algorithm are identical to the steps of basic GNG. The differences are the following:

- For the two closest nodes to the sample s and t, update the utility: $U_s = U_s + error_t - error_s$, where $error_t$ is the Euclidean-distance between node t and the sample.
- In the adaption step, U_n is multiplied by $(1 - \beta)$ for every node. The utility is small if a node is rarely a winner, or there is a neighbor close to it.
- In the standard GNG only those nodes are deleted that became isolated. In GNG-U, the node with the smallest utility i is removed differently. When a node is removed, GNG immediately inserts a new one near the unit producing the highest error (node j, with error $error_j$). We expect the newly inserted node to reduce this error. The higher $error_j$ is, the more likely we win by removing the worst node (node i), since the newly inserted node will be closer to the current sample. At the other hand, less useful nodes have to be removed. The rule: remove node i (having the lowest utilization) when $error_j/U_i > k$, i.e. when i is very useless, or when we win a lot by giving up the worst node and inserting a new one instead. Small k leads to fast tracking of changes. Large k leads to slow adaption. The value of parameter k typically used in the literature: $k = 1, \ldots, 200$.

Consequently this algorithm solves the problem of keeping dead nodes, but it forgets easily the past behaviour of the system, so it can't be used in dynamically changing observation space and it still uses many hyper-parameters.

2.3 Adaptive Incremental Neural Gas

To treat the problem of a relatively large number of hyper-parameters, the Adaptive Incremental Neural Gas algorithm (AING) offers a possible solution which has been published in [3] in 2013. The authors managed to define a completely parameter-free GNG variant where the decision of producing a new node for a new observation is based on a threshold, but there is a reasonably sophisticated heuristic to determine this threshold.

The general schema of AING can be expressed according to the following 3 cases. Let y_1 and y_2 respectively be the nearest and the second nearest nodes from a new sample x, such that $dist(y_1, x) < dist(y_2, x)$.

1. If x is *far enough* from y_1: a new node y_{new} is created at position of x.
2. If x is *close enough* to y_1 but *far enough* from y_2: a new node y_{new} is created at x, and linked to y_1 by a new edge.
3. If x is *close enough* to y_1 and *close enough* to y_2:
   - move y_1 and its neighbouring nodes towards x, i.e. modify their positions to be less distant from x.
   - increase the age of y_1's edges
   - link y_1 to y_2 by a new edge (reset its age to 0 if it already exists)
   - activate the neighbouring nodes of y_1
   - delete the old edges if any

A data-point x is considered far (respectively close) enough from a node y, if the distance between x and y is higher (respectively smaller) than a threshold T_y.

The paper [3] claims that the sum of distances from a node to its data-points can be computed incrementally, however, it seems to ignore the fact that nodes move while the algorithm is being executed. Our experimental implementation ignores this fact, too. The number of samples assigned to a node and the sum of the distances are stored for every node (we don't update these distances when a node moves), and the threshold is computed from these two quantities as follows:

- For every node, the threshold is the sum of the distance to the observations assigned to it plus the distances of the neighboring nodes, weighted by the number of observations assigned to them. Then, it is normalized (to make it an average).
- If a node has no samples assigned to it and has no neighbors, the threshold is half the distance to the closest node.

Each time a data-point is assigned to a winning node y_1, the age of edges emanating from this node is increased. Let n_{max} the maximum number of

data-points assigned to a node. A given edge is then considered "old" and thus removed if its age becomes higher than n_{max}

The authors propose also an interesting merging step to make the graph more compact. This merging step gets triggered when the number of nodes reaches an upper value. The more observations are assigned to a node and the higher its distance is from its closest neighbor, the smaller is the probability of being eliminated. Instead of relying on a hard rule based on a threshold, the merging step uses a more relaxed rule based on a probabilistic criterion. Saying that "a node y is far enough from its nearest node $\hat{y}$" is expressed as the probability that y will not be assigned to $\hat{y}$, according to the formula $P_{y,\hat{y}} = \frac{|X_y| \times dist(y,\hat{y})}{\kappa}$, where $\mid X_y \mid$ is the number of observations assigned to y node.

Parameter κ is the aggressivity of the merging process. It is defined as follows: Let d be the mean distance of all existing nodes to the center-of mass of the observed data-points. κ is incremented by $\kappa = \kappa + d$ each time the nodes need to be more condensed (at the last step of every merge iteration)

In conclusion this algorithm can handle also the dynamically changing behaviour of a system without forgetting the past behaviour (the merge step doesn't remove nodes with high number of assigned observations) and as hyper-parameters AING uses only the upper value parameter at the merging step avoiding any other hard to adjust hyper-parameters. On the other hand this algorithm still can't provide a density insensitive state-space representation and in our testing we sometimes encountered very long edges (at the beginning of the algorithm even those nodes get connected that are far away from each other due to initially lacking in observations), which might cause difficulties in clustering applications.

2.4 Summary

To evaluate the above mentioned three GNG algorithms we have prepared a benchmark. We generated uniformly distributed random samples from various shapes with well specified appearance and disappearance in time:

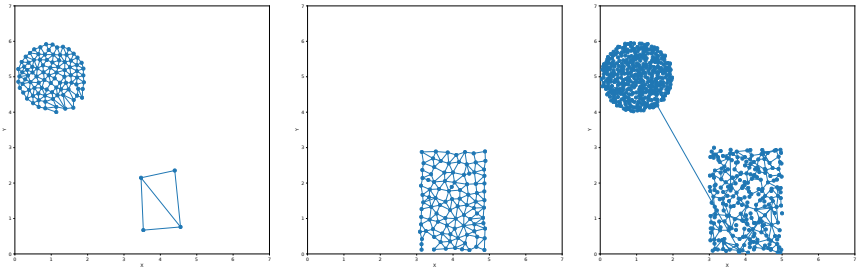
1. First we generate samples from a circle.
2. Then we made a sudden change, and generated data from a rectangle instead, that does not overlap with the circle.
3. Then we generated samples from an other circle again, and moved this circle continuously from the initial position to a final position.
4. Finally, we went back to the first circle and generated samples according to that.

With this benchmark we are able to study the behavior of the algorithms in the following cases:

- a sudden change occurs
- the support of the data is extended

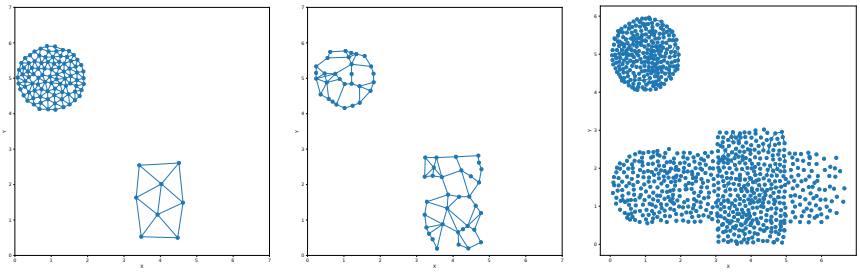
- continuous change of the distribution
- returning to old behavior.

We generated 100000 samples and the samples were divided to 200 batches. All three mentioned GNG algorithms are involved in the comparison: the traditional GNG algorithm, the GNG with utilization, and the adaptive incremental GNG algorithm. We have started all three algorithms from the same initial states, and followed their behavior after the training with the subsequent batches. Our experience is as follows. Till the first sudden change all three variants performed equally well. After the first phase, however, it become clear that the original GNG is unable to adapt to sudden changes: it could not grow new nodes, because it could not delete the old ones. GNG-U showed a perfect adaption, however, it forgot what it has learned before. AING represents a compromise: it distributed the nodes between the old and the new data area as well, representing samples from both region with acceptable accuracy (see Figure 1). These fundamental differences were observable all over the benchmarking process. Based on the end state, shown in Figure 2, we can conclude that the AING variant is able to preserve both older and newly learned data.



(a) Standard GNG (b) GNG with Utility (c) AING

Figure 1: The GNG variants right after the first sudden change of data distribution



(a) Standard GNG (b) GNG with Utility (c) AING

Figure 2: The GNG variants at the end of the training process

To quantify the difference between these GNG variants, we have introduced two metrics. The first one is the mean squared error of the nodes computed only for the last training batch. This metric is able to measure how quickly the procedure adapts to the change of the input distribution. The other metric is the mean squared error measured with all the training data (all batches) seen so far by the algorithms. The results are depicted by Figure 3 (Foreshadowing our solution, TiliNG. We will see, that TiliNG reaches similar accuray as AING, with significantly less nodes.) The numerical results confirm that GNG-U adapts very fast to sudden changes, but AING is not far behind. The plot showing the error with all batches seen so far is more interesting: the classical GNG is better than GNG-U, since it does not forget the past samples that fast. In this respect, the AING has the lowest error by far, especially when it's optional merging step is not used (like in our case), but we will see, that it uses orders of magnitudes more nodes.

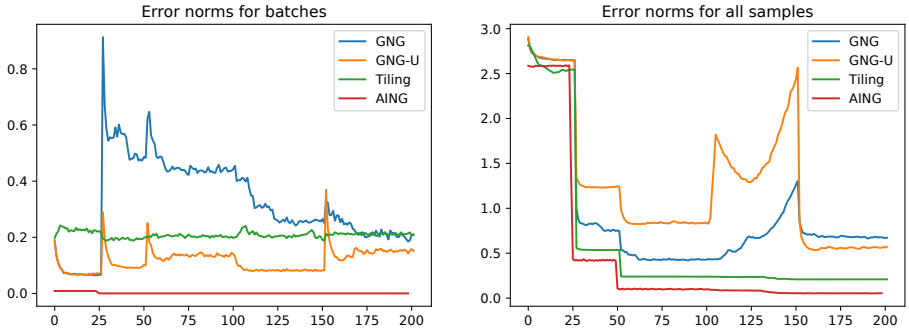


Figure 3: Batch and overall mean squared errors as the function of time

In conclusion taking into account the three problems of existing GNG algorithms mentioned in introduction our researches show that among these algorithms AING can handle the first problem the best. It can adapt to sudden changes without forgetting past behaviour, and it also uses only one hyper-parameter (upper-value, which is only used in an optional merging step). On the other hand neither of the three algorithms are prepared to handle the third problem: the density insensitive state space representation (see Figure 17).

In the next section we would like to present our suggested GNG algorithm which is able to handle sudden behavioural changes of the system, it has only one hyper-parameter and it can also provide a density insensitive state-space representation, independent on the observation's distribution so it won't ignore rare observations. In this algorithm we use a similar idea to the one mentioned in AING algorithm about applying the thresholds. In our GNG algorithm a radius parameter (R) will represent the threshold of the nodes.

3 TiliNG

The goal of our algorithm is to evenly cover the space of our observations with n-balls (n dimensional spheres) while trying to minimise overlaps between these n-balls.

A great disadvantage of all the GNG variations presented so far is their tendency to "overpopulate" dense areas. That is to say that most of the nodes are concentrated in parts of the state space, where most observations are made on the data set. Hence, the most common system states become over-represented and the less common system states become under-represented in the discrete model over time, especially if the number of nodes stored is fixed or limited. In some cases this can be considered an advantage, however there are applications, where it is an undesired behaviour.

Our proposed algorithm has a different goal than the GNG variants discussed in the literature (and presented here earlier). Beside modelling the system as realistic as possible we aim for a density insensitive state-space representation such that all visited parts of this space (even less popular states) can be identified. Therefore the gas has to grow without considering the density of samples in the state space, yet representing scarcely visited areas (i.e. enabling the exploration of the entire support of the sample space). Due to this density independent state-space representation this algorithm can be used efficiently in applications like anomaly detection.

As we mentioned in the introduction, we aim to eliminate as many hyper-parameters as possible. As a result we ended up using only one: R, the radius of the n-balls. It can be thought of as a precision or resolution parameter. Parameter R determines the area or volume around every node in which samples get assigned. Setting it low will result a more precise sample representation and an increase in the number of nodes.
In the next section we will also propose a procedure which can estimate the value of R as a function of the desired state-space resolution metric.

3.1 Steps of the algorithm

Our algorithm has three major steps, we are going to discuss them in the order they become relevant. The set of nodes is denoted by $\mathcal{N}$. The attributes of the node $y \in \mathcal{N}$ are as follows:

- Reference vector: *the position of the node (it is often referred to as y)*
- Sample counter: *the number of assigned samples so far*
- $\overline{y}$: *stores the upper bounds of the node's position, we do not let the node to go beyond these limits*
- $\underline{y}$: *stores the lower bounds of the node's position, we do not let the node to go beyond these limits*
- $\mathcal{X}$: *stores the other nodes within 2R distance for better performance*

3.1.1 Train

When a new sample s arrives, the algorithm searches for the closest, already existing node as follows: $y = \arg\min_z ||s - z||_2$. This node is referred to as best matching unit (BMU) hereinafter. In case $||s - y||_2 > R$, a new node is created at the sample's location, its sample counter gets initialized to 1 and its neighborhood set is initialized to $\mathcal{X} = \emptyset$. In case $||s - y||_2 \leq R$, the sample gets assigned to y, by incrementing its sample counter, and an adaptation and deletion phase starts.

3.1.2 Adaptation

If $\|s - y\|_2 \leq R$, no new node is created. Instead the position of y is adjusted and s is added to the samples assigned to y. This is done in accordance with the goals mentioned above. Let us denote the "local area" (the nodes within $2R$ distance, because only nodes closer than $2R$ can overlap with each other) of y by $\mathcal{X}$, more precisely, let

$$\mathcal{X} = \{x_i : \|y - x_i\|_2 \leq 2R\}.$$

Furthermore let us denote the extreme values of the coordinates of the samples assigned to y by $\underline{y}$ and $\overline{y}$ respectively. The adjusted position of y, $\hat{y}$ is given by minimizing its distance variance within its local area, i.e. by solving the optimization problem

$$\begin{aligned} & \hat{y} = \arg\min_y \sigma_y^2 \\ \text{such that} \quad & \hat{y} \geq \underline{y} \\ & \hat{y} \leq \overline{y} \\ \text{where} \quad & \sigma_y^2 = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \left(\|y - x\|_2 - \frac{1}{|\mathcal{X}|} \sum_{z \in \mathcal{X}} \|y - z\|_2 \right)^2 . \end{aligned} \tag{1}$$

We note here that there is no explicit solution for (1) in the general case. It is however possible to find solution with certain procedures, we will discuss these later in the article.

3.1.3 Deleting

Depending on the intended accuracy of the system's representation the number of nodes, especially in dense areas could prove to be unnecessarily high. Even with a lower resolution, (1) could have a solution, which overlaps in coverage with other nodes. Therefore after each adjustment step, the neighborhood of y has to be scanned for deletion. For a node to be selected for deletion, the set $\mathcal{B}_y = \{x_i : \|y - x_i\|_2 \leq R\}$ has to be non-empty. Then, since neighboring nodes' n-balls and the n-ball of y has an overlap, the relative intersecting volume for

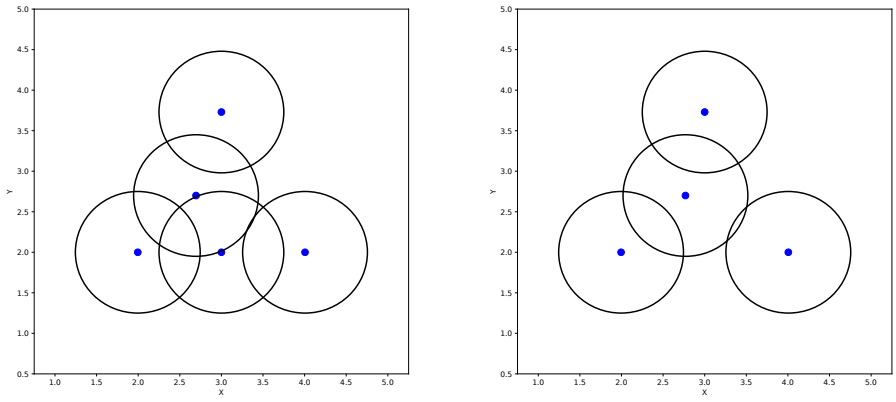
every node z in $\mathcal{B}_y$ (and for y) can be determined as

$$v_z = \sum_{x_i \in \mathcal{X}_z} I_{1-\left(\frac{\|z-x_i\|_2}{2R}\right)^2} \left(\frac{n+1}{2}, \frac{1}{2}\right),$$

where $I_t(a, b)$ is the incomplete beta function. Then $\hat{z} = \arg\max_z v_z$ is selected for deletion, as the loss of that node means the smallest loss in covered space.

On Figure 4 an example can be seen: the adaptation step moved the node in the middle closer to the center, which created a denser area. Thus the deletion step removed the most overlapping node, resulting in a more even coverage.

After the selection of the node to delete, we distribute its samples between its neighbors in proportion to their overlapping volume.



(a) Before deletion

(b) After deletion

Figure 4: Example of a deletion step

3.1.4 The role of the sample-counter

TiliNG does not rely on the number of samples assigned to a node, the goal of the algorithm is to cover the space of the samples' origin, as evenly as possible. This means, that depending on the application it can be completely ignored, thus saving computation time. We do believe however, that counting the number of samples assigned to the nodes can be useful in the following cases:

- They can be used to provide information on the density/distribution of the samples.
- Some data sets are incredibly noisy, there are applications, where it is desired to drop some of the nodes, that only have a few samples assigned.

An interesting utility measure for each node could be the product of the node's samples-counter and its distance to the closest node.

3.2 Approximate solution of adaptation

As we mentioned before there is no explicit solution for (1) but it is possible to efficiently approximate a solution.

3.2.1 Fix-point iteration with KKT

As it is described in (1) in the adaption step we also apply constraints for each dimension: the position of a node has to remain between the lower bound ($\underline{y}$) and the upper bound ($\overline{y}$), this way the nodes can not move to positions where no samples were observed. So we need to minimize the distance variance between the target node and its local area ($\mathcal{X}$) subject to the inequalities described in (1). For resolving this optimization problem we applied the Karush-Kuhn-Tucker(KKT) Theorem [4], which is able to handle the above mentioned problem.
We are trying to minimize the value of σ_y^2 (defined in (1)) subject to $\underline{y_k} \leq y_k \leq \overline{y_k}$ where $\underline{y_k}, y_k, \overline{y_k}$ are the respective values of $\underline{y}, y, \overline{y}$ in the k. dimension.
These constraints are equivalent to the following ones:

$$\begin{gathered} \underline{y_k} - y_k \leq 0 \\ \text{and} \\ y_k - \overline{y_k} \leq 0 \end{gathered} \tag{2}$$

So according to the KKT theorem the new function which needs to be minimized (primal problem) looks as follows:

$$\begin{aligned} L(y, \underline{y}, \overline{y}) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} (\|y - x\|_2^2) - \left(\frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \|y - x\|_2 \right)^2 \\ + \sum_{k=1}^{K} \alpha_k (\underline{y_k} - y_k) + \sum_{k=1}^{K} \beta_k (y_k - \overline{y_k}). \end{aligned} \tag{3}$$

where K is the number of dimension of input data, α_k and β_k are the Lagrange multipliers for each dimensions for both bounds ($1 \leq k \leq K, \alpha, \beta \geq 0$).

To minimize the value of L let's determine its derivative:

$$\begin{aligned} \frac{\partial L}{\partial y_n} = -\frac{2}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} x_n + 2y_n \\ - \frac{2}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \sqrt{\sum_{k=1}^{K} (x_k - y_k)^2} \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \frac{y_n - x_n}{\sqrt{\sum_{k=1}^{K} (x_k - y_k)^2}} - \alpha_n + \beta_n. \end{aligned} \tag{4}$$

We need to find y_n for which the value of (4) is 0.
Since expressing y_n from (4) is not easy using only standard methods, we will

express only an approximation of y_n using a fix point iteration (described in the next subsection).
Let $\hat{y_n}$ be the last value of y_n before executing the adaption step. This value can be considered as a real coefficient in the function and can be used for expressing value of y_n.

So the modified optimization problem is:

$$\begin{aligned} 0 = &-\frac{2}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} x_n + 2y_n \\ &- \frac{2}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \sqrt{\sum_{k=1}^{K}(x_k - \hat{y_k})^2}\frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \frac{\hat{y_n} - x_n}{\sqrt{\sum_{k=1}^{K}(x_k - \hat{y_k})^2}} - \alpha_n + \beta_n. \end{aligned} \tag{5}$$

For the sake of perspicuity let's introduce the following notations (as constant coefficients):

$$d_i = \sqrt{\sum_{k=1}^{K}(x_{k,i} - \hat{y_k})^2}, \text{ for } x_i \in \mathcal{X} \tag{6}$$

$$m_1 = \frac{1}{|\mathcal{X}|}\sum_{x_i\in\mathcal{X}} d_i \tag{7}$$

$$m_2 = \frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} (\|\hat{y} - x\|_2^2) \tag{8}$$

Then expressing y_n from $\frac{\partial L}{\partial y_n} = 0$ will result the following equation:

$$y_n = \frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} x_n + \frac{m_1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \frac{\hat{y_n} - x_n}{d_i} + \frac{\alpha_n}{2} - \frac{\beta_n}{2}. \tag{9}$$

Let c be the value of $\frac{1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} x_n + \frac{m_1}{|\mathcal{X}|}\sum_{x\in\mathcal{X}} \frac{\hat{y_n} - x_n}{d_i}$. More intuitively c would be the new position of y along dimension n if we had not applied the respective constraints of adoption.

If we replace y_n in (3) to the right side of (9) we will get the dual function $Q(\alpha, \beta)$. The value of this function depends only from α and β (not from y anymore). According to KKT theorem we need to maximize Q in function of α

and β:

$$\begin{aligned} Q(\alpha_n, \beta_n) &= m_2 - m_1^2 + \alpha_n(\underline{y_n} - c - \frac{\alpha_n}{2} + \frac{\beta_n}{2}) + \beta_n(c + \frac{\alpha_n}{2} - \frac{\beta_n}{2} - \overline{y_n}) \\ &= -\frac{\alpha_n^2}{2} + \alpha_n(\underline{y_n} - c) - \frac{\beta_n^2}{2} + \beta_n(c - \overline{y_n}) + \alpha_n\beta_n + m_2 - m_1^2, \\ &\text{subject to: } \alpha_n, \beta_n \geq 0. \end{aligned} \tag{10}$$

It can be seen that the maximum value of Q is the following:

$$\max\{Q|\alpha_n, \beta_n \geq 0\} = \begin{cases} \frac{1}{2}(c - \underline{y_n})^2 + m_2 - m_1^2, & \text{if } c \leq \underline{y_n} \leq \overline{y_n}. \\ m_2 - m_1^2, & \text{if } \underline{y_n} \leq c \leq \overline{y_n}. \\ \frac{1}{2}(c - \overline{y_n})^2 + m_2 - m_1^2, & \text{if } \underline{y_n} = \overline{y_n} \text{ or } \underline{y_n} \leq \overline{y_n} \leq c. \end{cases} \tag{11}$$

The α and β values belonging to the maximum value presented above are the following:

$$\arg\max_{\alpha_n, \beta_n}\{Q|\alpha_n, \beta_n \geq 0\} = \begin{cases} (\underline{y_n} - c; 0), & \text{if } c \leq \underline{y_n} \leq \overline{y_n}. \\ (0; 0), & \text{if } \underline{y_n} \leq c \leq \overline{y_n}. \\ (0; c - \overline{y_n}), & \text{if } \underline{y_n} = \overline{y_n} \text{ or } \underline{y_n} \leq \overline{y_n} \leq c. \end{cases} \tag{12}$$

This result shows that both constraints can not be active at the same time: if we maximize Q it is sure that either the value of α_n or β_n is equal to zero. In other words the results presented in (11) and (12) confirm the fact that the proposed new position by fix point iteration (c) can not be less than the lower bound ($\underline{y_n}$) and greater than the upper bound ($\overline{y_n}$) at the same time. If either of these two constraints is active, the solution of the optimization problem sets the optimal value of proposed BMU position between the predefined bounds.

Knowing the value of α_n and β_n the new value of y_n can be determined easily based on equation (9).

3.2.2 Fix-point iteration without constraints

As an efficient solution approximation for (1) we used an iterative fixed point method, which is based on the idea presented in Section 3.2.1 but instead of using any constraints in the objective function, it applies the correction presented in (14) for each dimension after $y(n+1)$ is determined as described in (13).

Let us use the notations

$$d_x(y) = \|y - x\|_2,$$

and

$$m_d(y) = \frac{1}{|\mathcal{X}|}\sum_{x \in \mathcal{X}} \|y - x\|_2.$$

The new position of y is then given iteratively by

$$y(n+1) = \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} x - \frac{1}{|\mathcal{X}|} \sum_{x \in \mathcal{X}} \frac{m_d(y(n))\,(x - y(n))}{d_x(y(n))}, \tag{13}$$

$$y(n+1) = \min\{\max\{y(n+1), \underline{y}\}, \overline{y}\}. \tag{14}$$

where n denotes the number of iterative steps taken. While (13) does not always converge to the optimum of (1) in the general case, but since we restrict the adjustment made to y by not allowing it to leave its bounds in any dimension ($\overline{y}$ and $\underline{y}$), it is more than enough to approximate it. This way the convergence is so quick, that it is reasonable to save computation time by taking only one step of (13). The more observation falls into the range of a node the closer it gets to the optimal position, see Figure 5.

We would like to point out, that introducing the bounds and using them in the adaptation step, the in-place preservation of outlier nodes, with only 1 sample assigned to them is ensured.

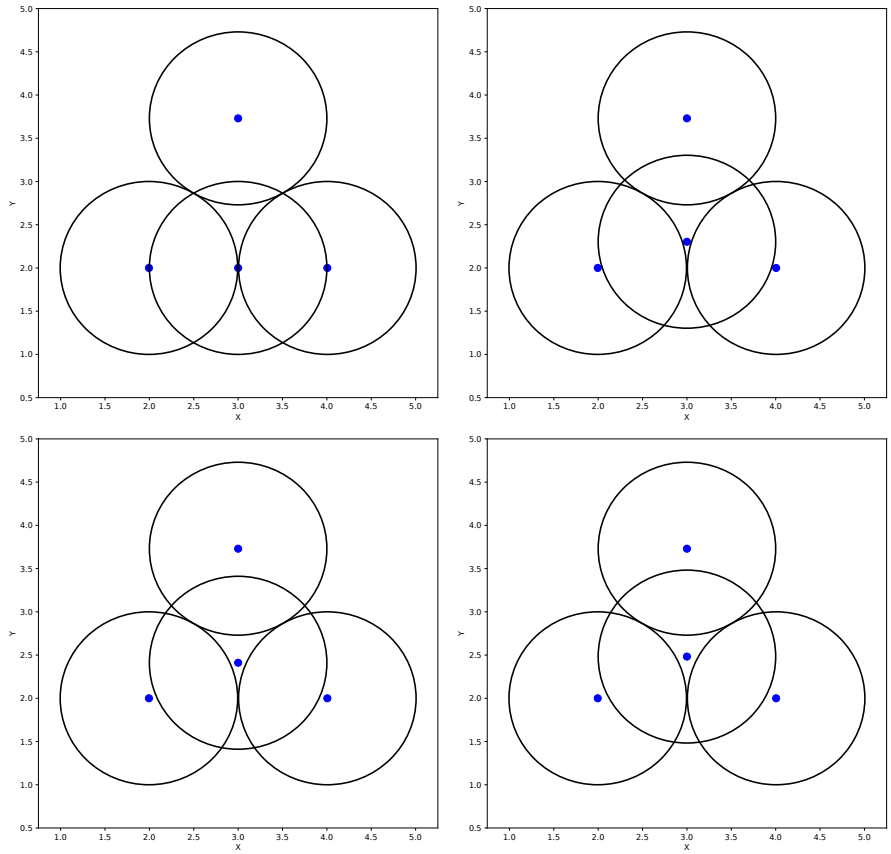


Figure 5: Node iterating to a better position

3.3 Graph structure

In order for the TiliNG algorithm to provide a neural gas structure that is consistent with the literature a graph form is needed. The nodes are given by the BMUs, however the edges still need to be defined. This also makes the model more structurally simple and raises the possibility of an implementation which has lower computation time by the use of the graph structure. The simplest solution to provide the graph is to connect every pair of nodes which are closer than $2R$ by an edge. However, the resulting graph is not topology preserving as defined in [5]. This problem can be alleviated in two ways:

- As the nodes given by TiliNG eventually represent states of the system, and a Markovian evolution is used to model the dynamics, the edges can be replaced according to the transitions in the model, making the original edge structure irrelevant.
- The graph can be made topology preserving by making sure that the

problematic edges are deleted after the final positions of the nodes are determined. For the sake of simplicity we demonstrate the topology correcting algorithm in only 2 dimension, but it can be used also for higher dimensional data sets. Let AB and CD be two intersecting edges as it is shown on Figure 6.

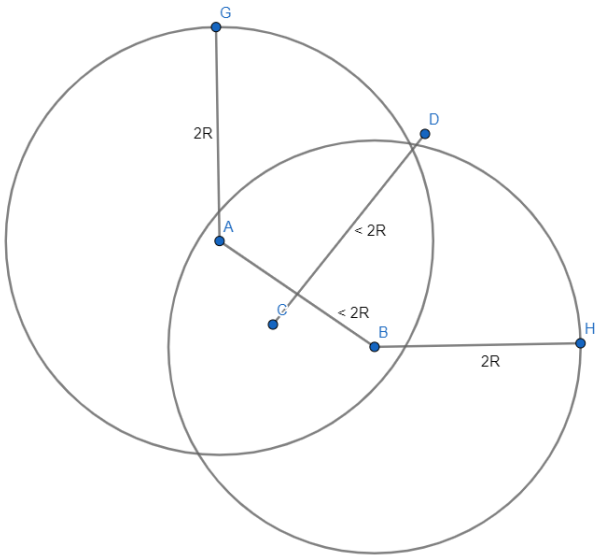


Figure 6: Example of intersecting edges

Knowing that $|AB| \leq 2R$ and $|CD| \leq 2R$, it can be easily seen that nodes A and B are connected to either C or D nodes. Therefore any edge intersecting with CD goes between the neighbors of C or D nodes. Based on these, the topology correcting algorithm iterates over the original edge list, which contains all node-pairs closer than $2R$ (let this be E) and creates the new edge list (E^*) the following way:

- Determines the possible intersecting edges from E (based on the train of thoughts described above) of the currently inspected edge. Let this set be I.
- If the inspected edge (e) does not intersect with any edge from I then e is inserted into E^*.
- If the inspected edge (e) does intersect with an edge e_i, where $e_i \in I$ and $e_i \in E^*$ then e will not be inserted into E^*.
- If the inspected edge (e) does intersect with any edge e_i, where $e_i \in I$ and $e_i \notin E^*$ then e will be inserted into E^* and all e_i edges will be removed form E and excluded from further inspections.

After applying these steps E^* will contain the topological correct edge list. An example of the correcting algorithm's result can be seen on Figure 7. We ran the Tiling algorithm on our benchmark data set presented in Section 2.4.

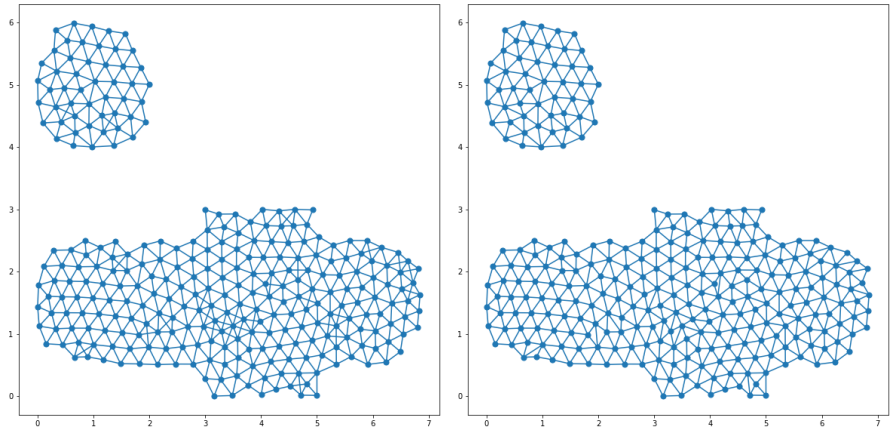


Figure 7: Graph structure before and after running topology correcting algorithm

3.4 Efficiency tricks, Implementation

Python is a relatively slow language, yet it has an enormous toolbar for data-science, tools we wanted to use in our testings. To reach a better performance we wrote TiliNG in C++ and made a Python interface using Boost [6].

We wanted to make our algorithm as fast as possible, so we tried to avoid as many costly operations as possible. Since TiliNG spends most of the time calculating the distances between two points in the n dimensional space, we store not only the neighbors of a node, but also their distances. Nowadays multi-core processors are so common it is a good idea to exploit the additional computational power, this is why we prepared TiliNG to be able to run thread-safely on multi-core processors, using OpenMP [7].

After this we took a look at our Euclidean-distance calculation. Notice that TiliNG does not require the real distances between points. Their squares are more than enough, this way we save a costly square root calculation. The only thing we need to keep in mind, that the numbers we compare to the distance must be squared as well. This however is not a problem, since multiplying numbers are much easier than finding their root.

Further improvements can be reached by implementing a better search and store algorithm (currently the nodes are stored in a vector and the search is linear).

3.4.1 Spatial search

The finding of the closest node to a given point is a crucial part in every GNG variant so it makes sense to try and minimise the resources spent on searching. For smaller dimensions the well known R-Trees (preferably R*-Trees) can be more than enough, but for higher dimensions X-Trees as written in [8] usually perform better (up to two orders of magnitude).

3.4.2 Pseudo-code

Below we include the pseudo-code of the algorithm.

Algorithm 1 TiliNG

```
1: Initialization by making a new node at the first sample
2: while some data-point remain unprocessed do
3:     get next data-point x
4:     let y be the closest node to x
5:     let d be the euclidean distance between y and x
6:     if d > R then
7:         make new node at x
8:     else
9:         ȳ = max(ȳ, x) for each dim
10:        y̲ = max(y̲, x) for each dim
11:        ȳ → adapt
12:        y → increment sample counter
13:        let neigh be the neighborhood of y with y included
14:        while ∃z ∈ neigh with dense area do
15:            ẑ = arg max_z v_z where v_z is defined in (3.1.3)
16:            distribute the samples of ẑ as it is mentioned in (3.1.3)
17:            remove ẑ from the nodes
```

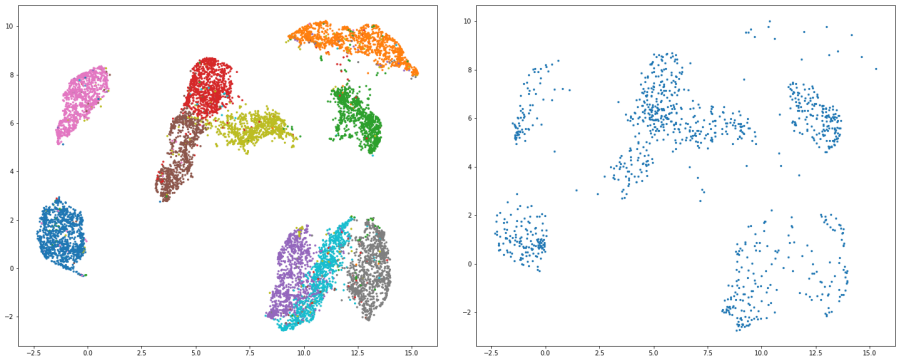
3.5 Scalability and complexity

TiliNG (with efficent search implementation, for example R*-Tree) has the potential to reach a better complexity level than $\mathcal{O}(n^2)$. Let us denote n to the total number of samples, in this case the total number of nodes will be $\leq n$. On average R*-Trees provides an $\mathcal{O}(log(k))$ complexity for search and insertion, where k is the current number of nodes. n can be considered an upper estimate for k, since $k \leq n$ must be true (in the practical use-case $k \ll n$). To each sample, the closest node must be found exactly once, which means, that the search (and possible node insertion) must be done n times. So TiliNG's complexity on average is: $\mathcal{O}(n \log(n))$.

The low complexity of the algorithm makes it viable for large data sets. To confirm this we run the algorithm on the whole MNIST [9] data set, which consists of gray-scale pictures of handwritten digits (0-9), each with 28*28 pixels, for a total of 784 dimension. The results can be seen on Table 1 . The 2D projection (we used UMAP [10] of the constructed nodes can be seen on 8 . Unfortunately the 2D projection could not keep TiliNG's even coverage, but we can see that the nodes perfectly aligning to the samples.

Number of nodes	R	Time
13	11	6.055 s
36	10	8.088 s
1054	8	354.8 s

Table 1: TiliNG's performance on the MNIST data set



(a) MNIST data set (b) Constructed nodes

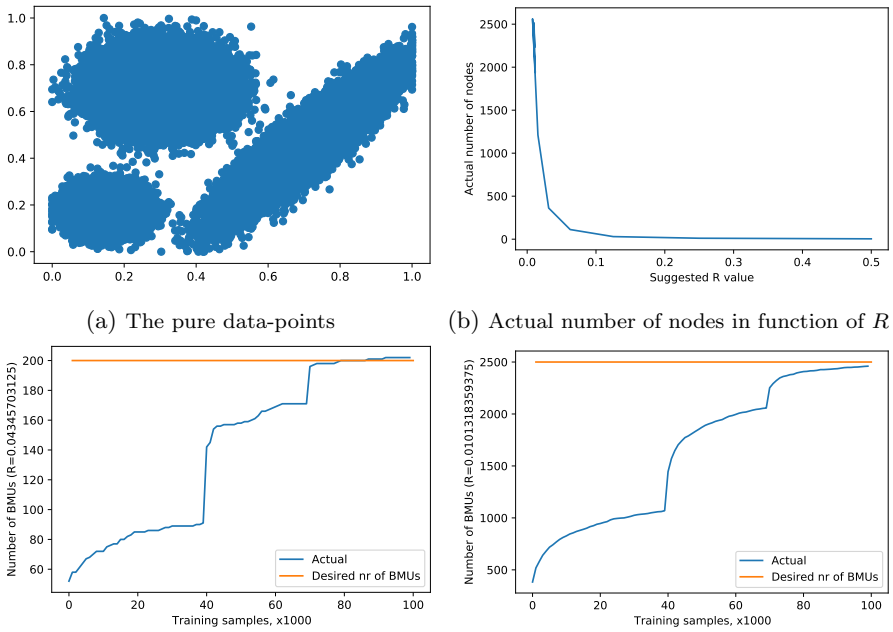
Figure 8: The 2D projection of the MNIST data set and the 1054 nodes

4 Proposed approximation of parameter R

TiliNG was designed to represent the space with a certain precision: R. In some cases the fixed number of nodes might be more desired than the option to choose the resolution. It is easy to see that there is a correlation between the resulting number of nodes and R. This raises the question: how to choose R to end up with a certain number of discrete states. To answer this we would like to propose three ways to approximate the number of nodes using R.

4.1 Trial and error

The simplest and probably the least effective way to do this is to get a portion of all the observations (it has to be large enough to cover the whole space with a high enough probability) and then run parallel TiliNG on it with constant number of different R settings. At the end of each execution of the algorithm count the nodes. If in case of some R_1 value the number of nodes starts to converge in range of the desired node number then we choose R_1 as the parameter of TiliNG. If the data-set was large enough it provides a good approximation for the whole data-set as well. On Figure 9 we can see how TiliNG reaches almost exactly the desired number of nodes, when it is run with the proposed R parameter. Note, that each bigger step on the visuals are corresponding to the arrival of one of the 3 clusters showcased on Figure 9a . On the Figure 9b it can be seen how the actual number of nodes variates in function of the R values suggested by trial and error method when the desired number of nodes is 2500. The prcedure starts guessing with larger R values (e.g. 0.5, 0.25) and at the end it converges at 0.01. Around this R value the actual number of nodes will be close to 2500 after running the TiliNG algorithm on the whole data set. In this test case the algorithm needed only 14 trials to find the optimal R value.



(a) The pure data-points

(b) Actual number of nodes in function of R

(c) Number of desired nodes: 1000

(d) Number of desired nodes: 2500

Figure 9: Number of nodes by different R values along training process

Desired num. of states	$\hat{R}_{bisect}$	$n(\hat{R}_{bisect})$
200	1.016	206
1000	0.53	1073
2500	0.3712	2596

Table 2: Estimation of parameter R based on the Trial and error method

We would like to note here that this solution is rather slow and resource-demanding. Neither of those are desirable properties. Considering this step only prepares for the real run of TiliNG, it should be fast and efficient. Keeping these in mind we would like to suggest a more elegant and faster procedure.

4.2 Multi-normal distribution

For this solution to work, it is assumed that our observations follow a normal distribution. Let $\hat{R}$ be the optimal resolution for the intended number of nodes (n). In this case let $n(R)$ be the functional relationship between the resolution and the number of nodes. In this method firstly we provide bounds for $\hat{R}$ based on analytical arguments. Namely, if we fit a multivariate Gaussian-distribution to the data, then we can compute a bounding ellipsoid which provides a convex hull for the positions of the nodes. The semi-axes of the ellipsoid are given by

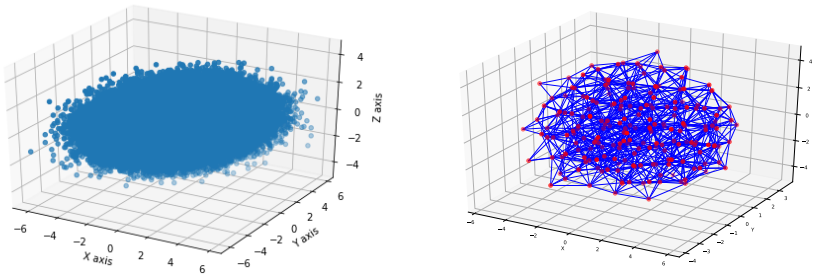
eigenvalues of the covariance matrix of the distribution (which we note here by λ_i), which can be easily computed from the data. The bounds are then provided for n(R), by dividing the volume of the ellipsoid with the volume of the N-dimensional spheres that the nodes represent as

$$\prod_{i=1}^{N} \frac{\lambda_i + R}{R} \leq n(R) \leq \prod_{i=1}^{N} \frac{\lambda_i + R/2}{R/2}, \tag{15}$$

where the lower bound assumes non-overlapping spheres, and the upper bound assumes nodes which are exactly in R distance from each other. The bounds for $\hat{R}$ can be given by inverting (15). The results of the approximation on our benchmark data set can be seen on 3. It also showcases, that although our observations did not follow a normal distribution, this method can still produce usable guesses.

Desired num. of states	$\hat{R}_{approx}$	$n(\hat{R}_{approx})$
200	0.2719	160
1000	0.1160	793
2500	0.0723	1890

Table 3: Estimation of parameter R based on the multi-normal method



(a) 3D multi-normal distribution

(b) Tiling result with R = 1.016

Figure 10: Raw data and TiliNG node locations on samples from multi-normal distribution

The mean of the obtained bounds is a relatively good approximation of the optimal resolution. On Figure 11 we tested the method on our benchmark and a few other data sets. It shows greatly how the method can fail on special, artificial data sets, that may not follow a normal distribution. During this estimation we have multiplied the eigenvalues with 3, because in a normal distribution most of the samples are included in the interval $[\mu - 3 * \sigma; \mu + 3 * \sigma]$. The other data sets do not fill their approximated ellipsoid completely (as they do not follow a normal distribution), thus the more they differ from a normal distribution, the

more the procedure overestimates the resulting number of nodes.

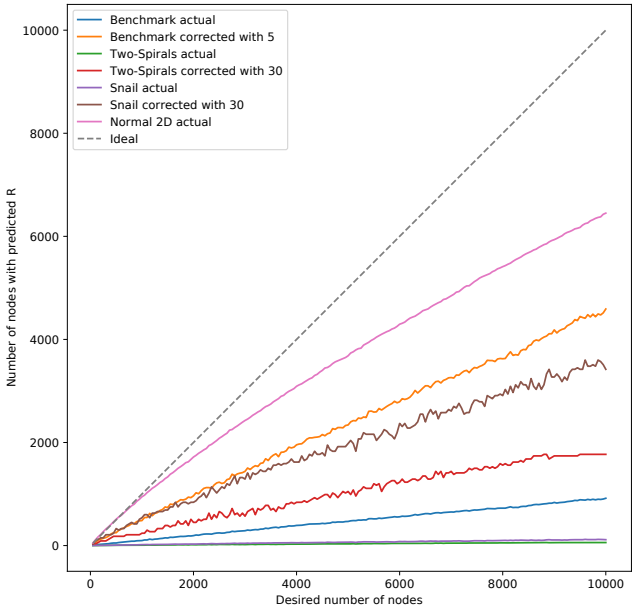


Figure 11: The predicted in relation to the desired number of nodes

4.3 Combined method

As a third and most efficient method we suggest combining the above mentioned two methods in the following way:

1. Firstly we provide bounds for $\hat{R}$ using the second method, which is based on the Multi-normal distribution. Let R_l and R_u be respectively the lower bound and upper bound (obtained from (15)) of the optimal resolution $\hat{R}$.

2. As a second step we apply the Trial and error method for different R settings in the interval defined by the lower and upper bounds of $\hat{R}$. For this step we run tiling only for one batch of data for all chosen R values. Each version produces a certain number of nodes. As a final step we choose the R value belonging to the version with the closest number of nodes to the desired number.

To evaluate the above mentioned we have used the same benchmark data set as the one mentioned in 2.4. A few numerical results can be seen in Table 4 below:

Desired num. of states	$\hat{R}_{lower}$	$\hat{R}_{upper}$	$\hat{R}_{bisect}$	$n(\hat{R}_{bisect})$
200	0.1812	0.3625	0.2447	201
1000	0.07735	0.1547	0.1044	950
2500	0.0482	0.0964	0.0626	2481

Table 4: Estimation of parameter R based on the combined method

On Figure 12 we can see that the number of nodes converges towards the desired number, when TiliNG is run with the parameter R proposed by the combined method.

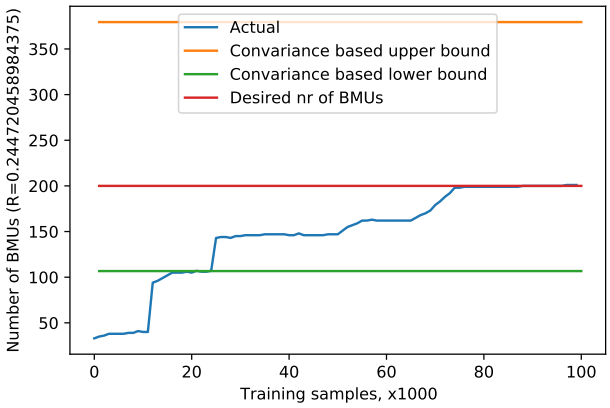


Figure 12: Number of nodes by proposed R value along training process (desired number of nodes = 200)

5 Evaluation and results

In this section we would like to present the results of TiliNG on our benchmark data sets and comparisons between the other GNG variant and our proposed algorithm.

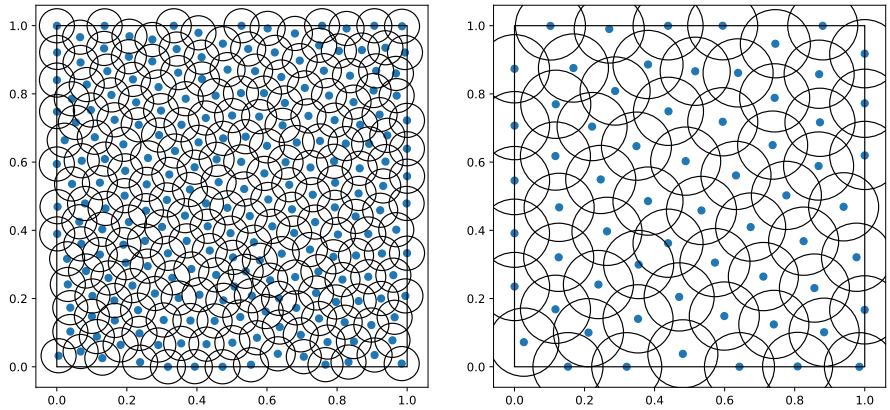
5.1 Evaluation of TiliNG

Firstly we would like to show how TiliNG performs on famous 2 and 3 dimensional data sets.

5.1.1 2D data sets

Figure 13 shows the state representations created by TiliNG with different R parameters on a data set sampled uniformly from a unit square. It can be seen that TiliNG managed to create an evenly distributed state-space representation and in order to minimize distance variance TiliNG also placed nodes close to

the borders of the data set. On Figure 14 it can be seen that the algorithm performed equally well on the Fibonacci snail data set having the parameter $R = 0.4$. The circles as radius indicators on the right side of Figure 14 show that the distance of each pair of nodes is an optimal value between R and $2R$.



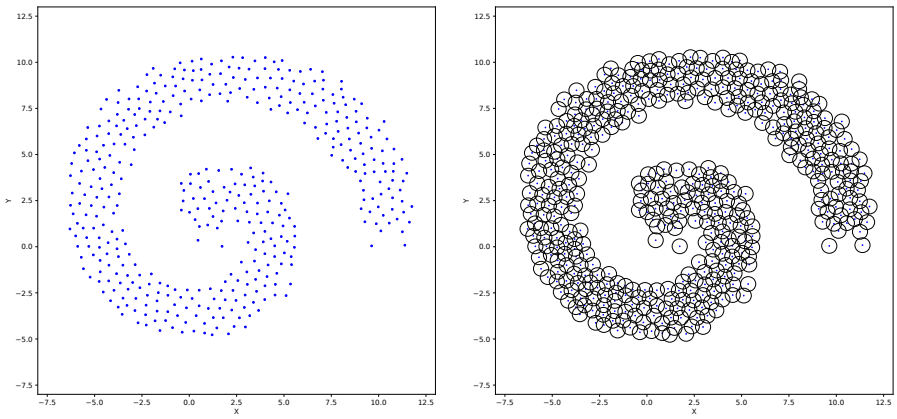
(a) R = 0.05

(b) R = 0.1

(c) R = 0.2

(d) R = 0.3

Figure 13: Tiling neural gas BMU locations on unit square with different R parameters

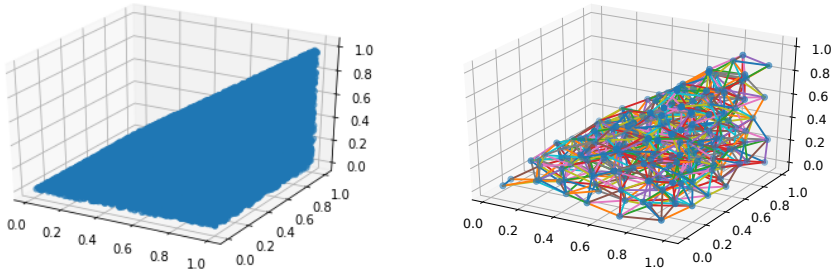


(a) R = 0.4 (b) R = 0.4

Figure 14: Tiling neural gas BMU locations on Fibonacci snail with and without radius indicators.

5.1.2 3D data set

Figure 15 shows a 3D state-space representation created by TiliNG. The data set for this test case was uniformly sampled from a 3D "Wedge-formed" structure (left side of the figure). The TiliNG result is presented as a graph with colored edges for better visibility (As mentoned in previous sections, two nodes are connected by an edge if they are closer than $2R$).



(a) The data samples (b) TiliNG's result

Figure 15: Wedge data set - 100000 samples, $R = 0.12$

5.2 Comparison with GNG variants

As we mentioned in Section 3 beside modelling the target system as realistic as possible, TiliNG aims for a density insensitive state-space representation. Furthermore, it was an important aspect for us to create an algorithm which can adapt to changes in the target system without forgetting its past behaviours. Therefore in the following subsections we would like to present comparison results between TiliNG and other GNG variants (Standard GNG, GNG with Utility, AING) based on these two aspects.

5.2.1 Uniform density

To compare the performance of the above mentioned GNG algorithms we ran all of them on a data set sampled from a 2D normal distribution as shown on Figure 16. The state representations produced by the respective GNG algorithms are presented on Figure 17. It can be seen that Standard GNG and GNG with Utility only represent the dense area of the data set, they totally ignore outliers which can be a problem in case of some application, for example anomaly detection (see 6.2). In contrast to Standard GNG and GNG with Utility, AING does not ignore outliers, but it highly over represents the dense area, which is disadvantageous when the maximum number of usable nodes is fixed. It can be seen that TiliNG gives solution to both of the problems mentioned above: it keeps the outliers easily identifiable and does not over represents the dense area either. So the result is a evenly distributed realistic state-space representation.

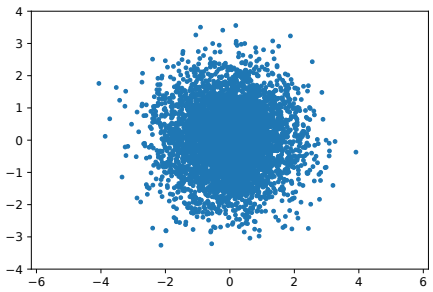
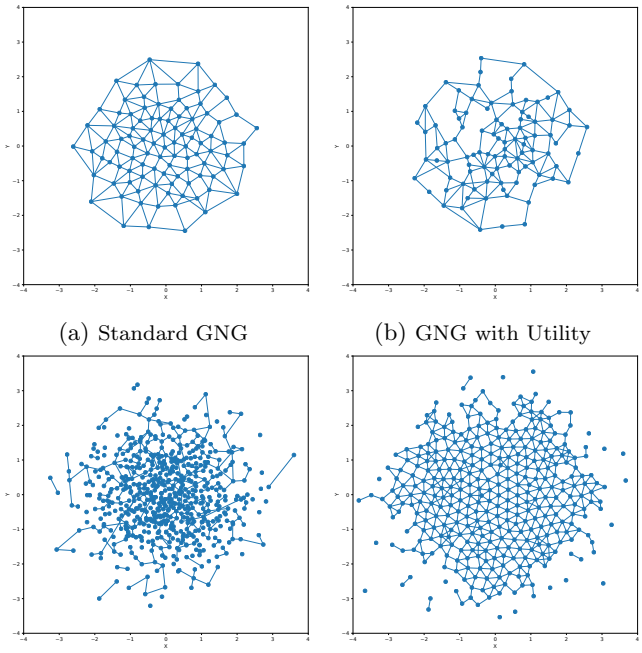


Figure 16: 2D normal distribution data set



(a) Standard GNG (b) GNG with Utility

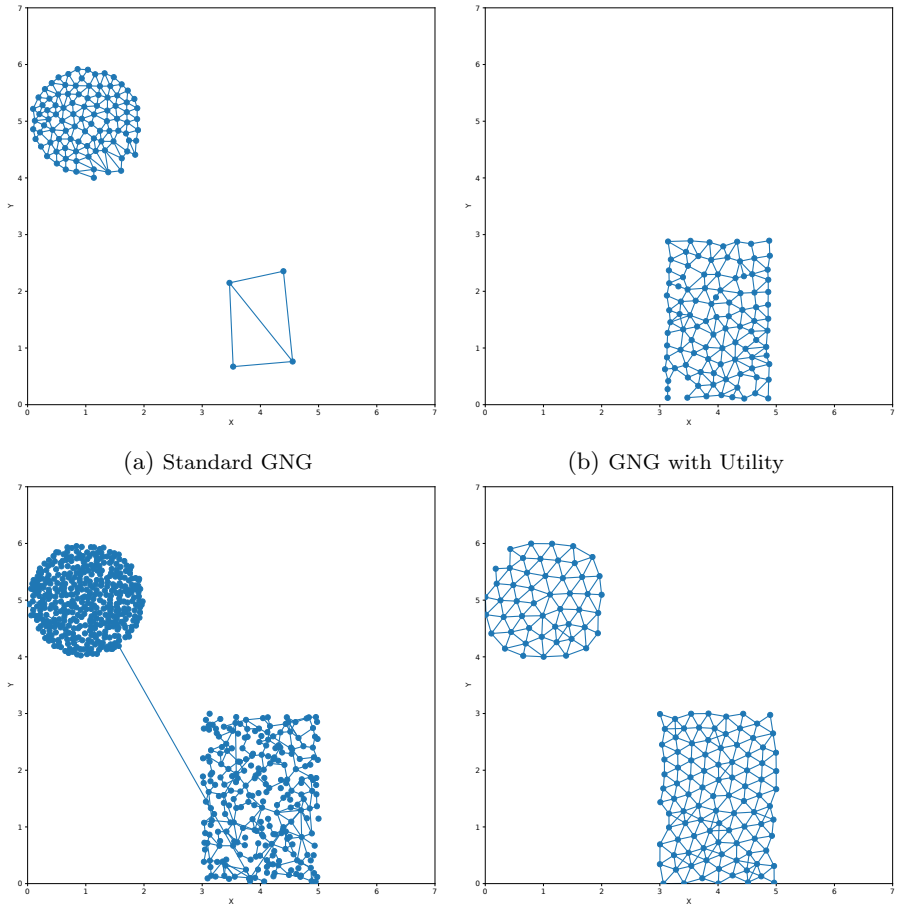
(c) AING (d) Tiling

Figure 17: The four GNG variants on 2D normal data distribution

5.2.2 Remembering past behaviours of a system

When it comes to compact data representation it is crucial that all of our samples are actually represented, otherwise we can not draw meaningful conclusions and discover hidden connection, because parts of our samples would be missing. Our representation also must be able to adapt to changes in the samples' behaviour, otherwise we arrive at the same problem: subsets of our samples will not be represented. We designed TiliNG with this requirement in mind, and now we would like to show how our method compares to other GNGs. For this purpose we use the data set introduced in 2.4 .

As we can see on Figure 18 the standard GNG tried, but failed to completely adapt, while the GNG with Utility adapted too perfectly, forgetting the system's previous state. TiliNG and AING perfectly adapted to the sudden change of data, but TiliNG covered the space with orders of magnitude less node and does it with an even density. The latter can not be said about AING, its circle representation is clearly denser.



(a) Standard GNG

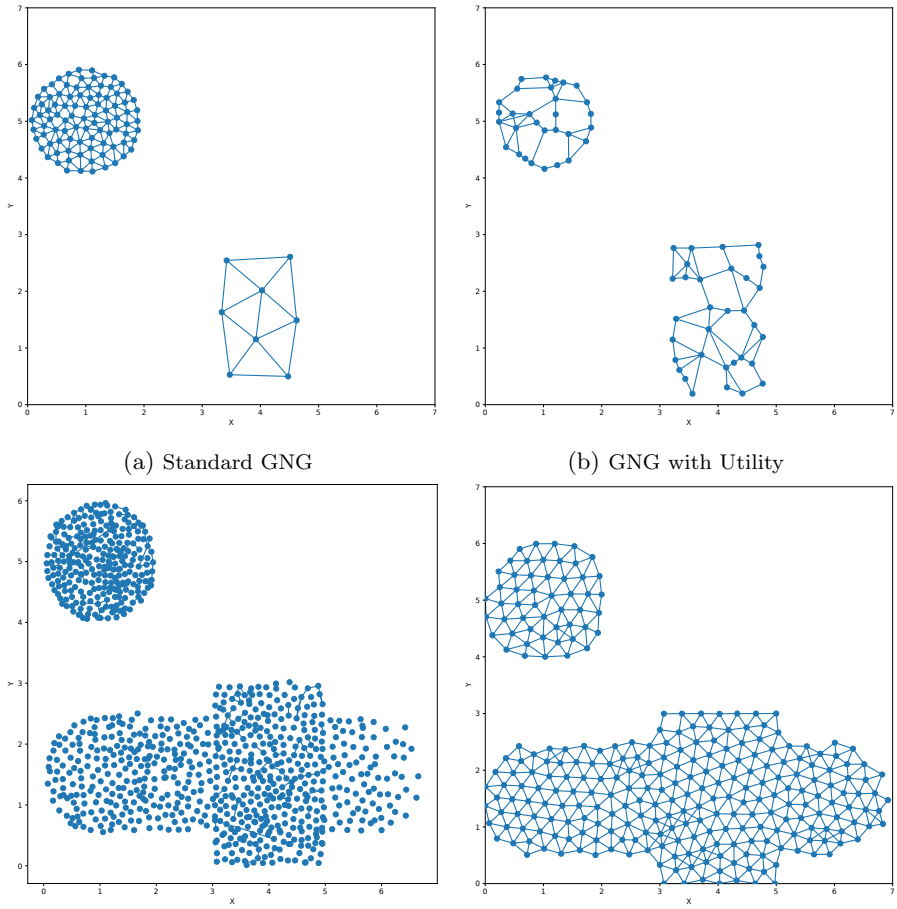
(b) GNG with Utility

(c) AING

(d) Tiling

Figure 18: The four GNG variants right after the first sudden change of data distribution

On Figure 19 we can see the four GNG variants at the end of the training process. The Standard GNG and the GNG with Utility failed to remember the continuous change in data, leaving a huge subset of samples unrepresented. Again, TiliNG and AING conserved the past states of the system perfectly, but TiliNG did so with orders of magnitudes less nodes and resulted in an even state representation.



(a) Standard GNG

(b) GNG with Utility

(c) AING

(d) Tiling

Figure 19: The four GNG variants at the end of the training process

On Figure 20 we can see the mean squared errors over time for batches of samples and for all the samples. The former is capable of measuring how quickly the algorithms adapt to changes, while the latter can measure the overall performance (note, that in this case the errors were always computed on all of the samples, regardless of whether the algorithm has seen them or not, thus the sudden drops in errors). As we can see, the standard GNG shows a decrease in errors in batches, it seems to be improving, but looking at the ascending part of the overall errors, it is evident that it forgot the previous behaviours of the system. Similar observations can be made in the case of GNG with Utility. It shows some spikes on the batch errors, these correspond to the changes in data. A look at its error on all samples reveals that it could not remember the past states of the system either. TiliNG and AING has pretty much constant errors (apart from the drops), and AING seems to be performing better, but

keep in mind, that TiliNG uses a fraction of the nodes used by AING. TiliNG's even coverage also manifests in the slightly higher errors, because it does not overpopulate dense areas, instead it moves nodes to achieve a smaller distance-variance with its surrounding nodes.

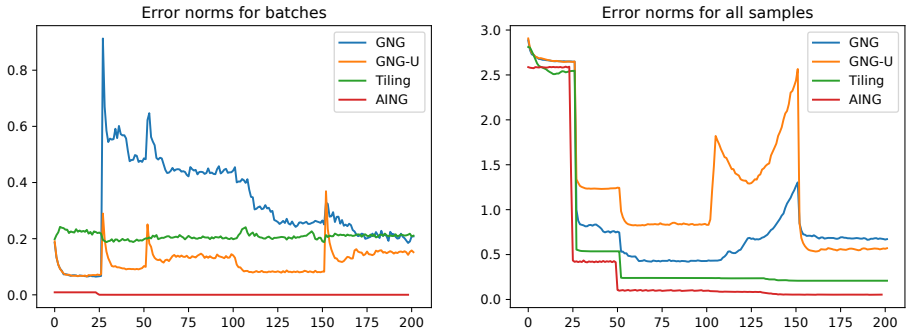


Figure 20: Batch and overall mean squared errors as the function of time

6 Applications

In this section we would like to present a few applications, where TiliNG could prove to be useful.

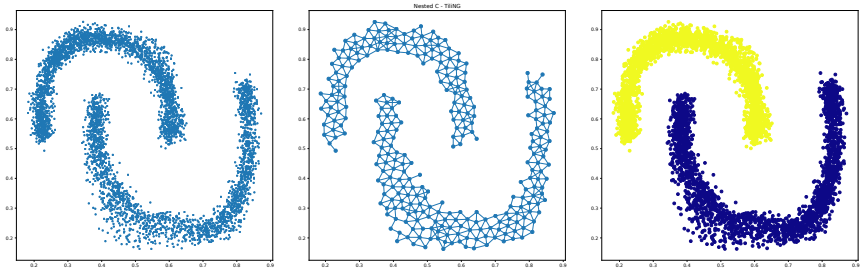
6.1 Clustering

Clustering, especially in high dimensions can be a very hard problem and there are many applications, where the resulting clusters at the end of an algorithm can not be validated. To get some kind of validation one might try different clustering algorithms, to compare their results and get a sense of how confident and reliable their clusters are. However: these algorithms can become relatively slow on large and/or high dimensional data-sets. We beleive that TiliNG may be able to help in these applications.

6.1.1 TiliNG as a useful tool for clustering algorithms

The main idea is that we do not need to run necessarily these algorithms on the whole data-sets. One might wonder: if only there would be an algorithm that can help reduce the number of observations, while keeping the important features. Well, this is exactly what TiliNG can help with, thus speeding up the clustering procedures. We note here, that TiliNG was designed to be as density insensitive as possible so it might not be an ideal choice for density-based clustering algorithms, it can however produce edges, thus allowing the use of graph-based clustering algorithms. Combining TiliNG with any component searching algorithm results a new clustering procedure. To demonstrate this we used the NetworkX [11] Python module's connected_components procedure

on the graph structure produced by TiliNG. The combination of these two algorithms was able to effectively determine the clusters on both our benchmark data sets (Figure 21 and 22).

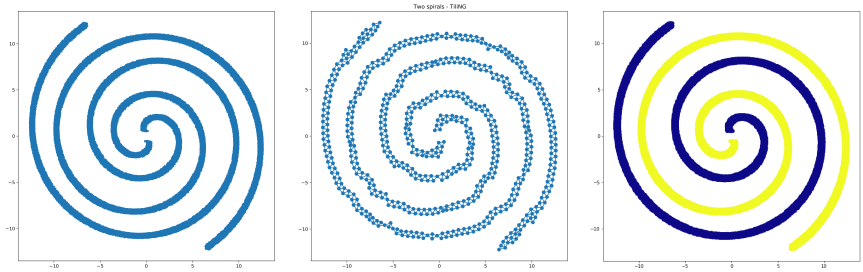


(a) Nested C - Raw data points

(b) Tiling result on Nested C dataset

(c) Tiling with component searching as clustering algorithm

Figure 21: Tiling neural gas clustering on nesed C dataset



(a) Two spirals - Raw data points

(b) Tiling result

(c) Tiling with component searching as clustering algorithm

Figure 22: Tiling neural gas clustering two nested spirals

6.1.2 Comparison with well-know clustering algorithms

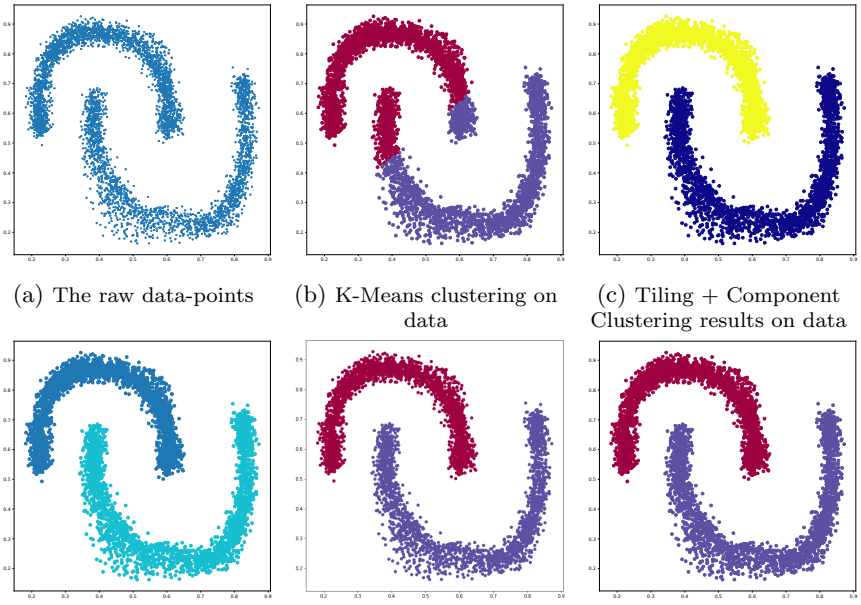
In this section we would like present a few famous clustering algorithms' result on the previously mentioned "Nested C" (or interleaving half circles), "Two twisted spirals" and also the high dimensional MNIST [9] data set. The inspected clustering algorithms are: K-Means [12], DBSCAN [13] and HDBSCAN [14]. We compared TiliNG's clustering performance with these algorithms based on correctness of the created clusters and execution time.

Figure 23 and Figure 24 shows the results of the above mentioned procedures on our benchmark data sets. It can be seen that K-Means could not find the two distinct part of the interleaving half circles and twisted spirals. K-Means tries to place a predefined number of centroid in the centre of clusters, therefore this

algorithm doesn't perform well on data sets which are not linearly separable. In contrast to K-Means, DBSCAN and HDBSCAN clustered the data sets as it was expected (in case of DBSCAN we determined by a few trials the best epsilon parameter which suits these data sets). In these experiments we examined how TiliNG performs together with DBSCAN in clustering tasks applying the following steps consecutively:

1. Running TiliNG on the raw datasets.
2. Running DBSCAN on nodes produced by TiliNG
3. Assigning a cluster index to each data points based on its BMU.

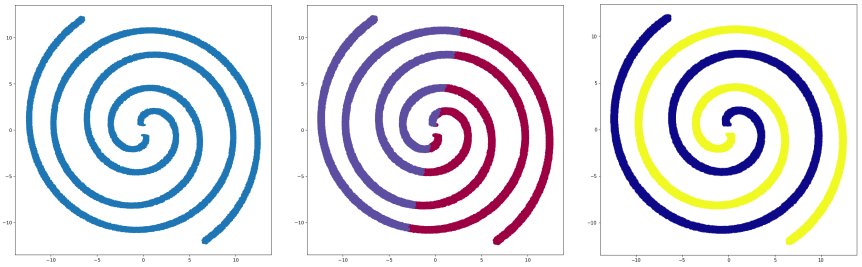
DBSCAN is a density based clustering method so as we expected it performed well on the evenly distributed state-space, generated by TiliNG. The execution time was far better in this case than pure DBSCAN's execution time. Furthermore TiliNG can be useful in determining the epsilon parameter of DBSCAN: the optimal value of epsilon is highly correlated to the density of the data set and it can be seen that the value of TiliNG's R parameter multiplied by two is (as a rule of thumb) a good choice for epsilon (R can be estimated by one of the methods presented in 4).



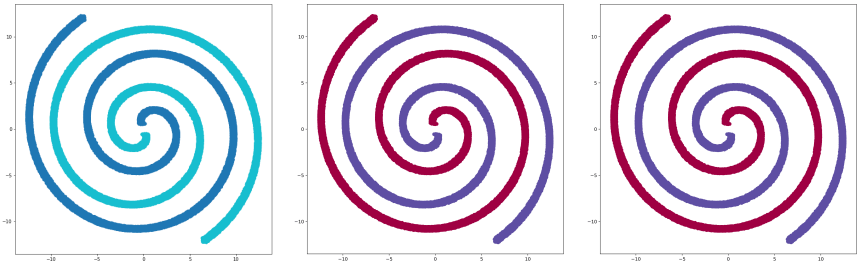
(a) The raw data-points (b) K-Means clustering on data (c) Tiling + Component Clustering results on data

(d) TiliNG and DBSCAN result on the data (e) DBSCAN clustering on data (epsilon = 0.04) (f) HDBSCAN clustering on data

Figure 23: Results of the clustering algorithms on the nested C data set



(a) The pure data-points (b) K-Means clustering on data (c) Tiling + Component Clustering results on data



(d) TiliNG and DBSCAN result on the data (e) DBSCAN clustering on data (f) HDBSCAN clustering on data

Figure 24: Results of the clustering algorithms on the twisted spirals data set

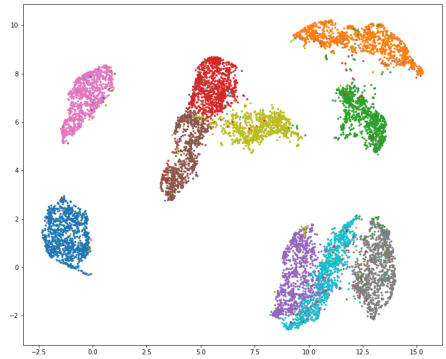
On Table 5 we can see the result of comparing the execution time of the examined clustering algorithms on our two twisted spirals data set. (Unfortunately the Scikit-learn [15] implementation of DBSCAN runs out of memory after a few minutes, on a machine with 16GB of RAM.) It seems that clustering with TiliNG greatly speeds up the process. We have achieved similar results on our other data sets.

Number of data samples	Tiling and ComponentCluster	Tiling and DBSCAN	DBSCAN	HDBSCAN
$1.0e^5$	1.39 s	1.3939 s	2.11 s	3.81 s
$1.0e^6$	14.154 s	14.158 s	$>$ 120 s	97.887 s

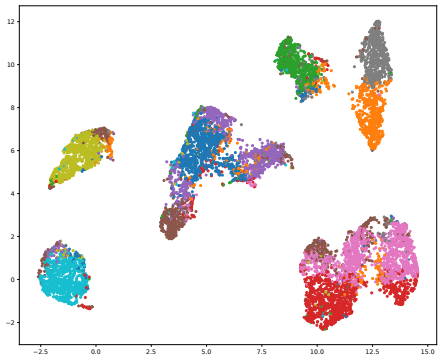
Table 5: Execution times on "two twisted spiral" dataset with different clustering algorithms

On the high dimensional, MNIST dataset (handwritten numbers, 28x28 greyscale pictures) we have experienced mixed results, which emphasises how hard clustering in high dimensions is (mainly because of the curse of dimensionality). Transforming to $2D$ was done using UMAP [10] . We have colored the data-points according to the number they represent, this would be the ideal clustering (Figure 25/(a)). K-Means performed surprisingly well, HDBSCAN rather poorly, while TiliNG represents the middle way: it concentrated the sam-

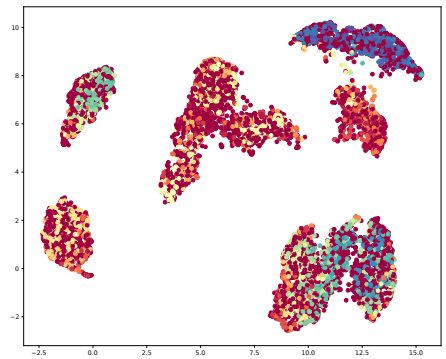
ples of few cluster to the same space. Tiling was run with an approximated R parameter, that would most likely result in 10 nodes representing the clusters. In all of these cases the clustering was done in the high dimensional space, UMAP was only used to project the results to 2D.



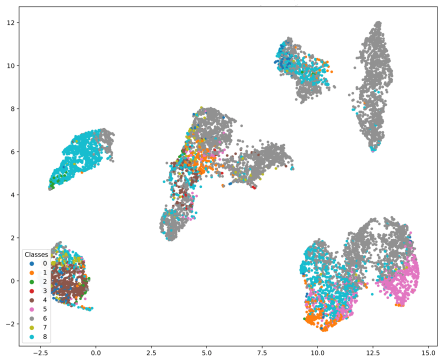
(a) The raw data-points, the ideal clustering



(b) K-Means clustering in high dimension



(c) HDBSCAN clustering in high dimension

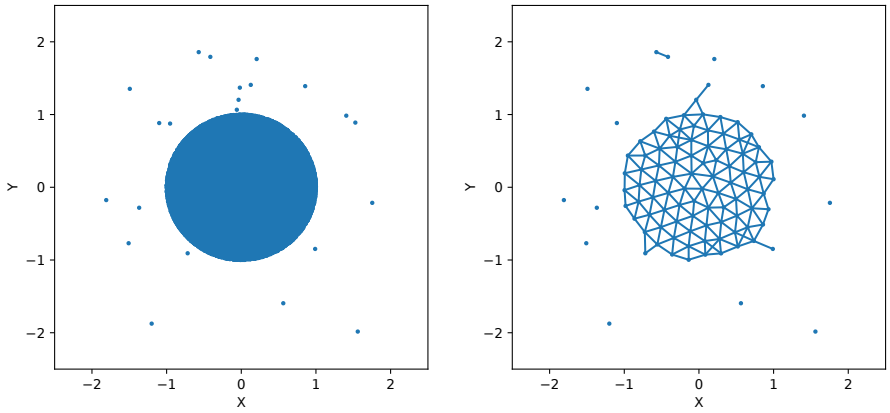


(d) Tiling clustering in high dimension

Figure 25: Clustering algorithms in high dimension

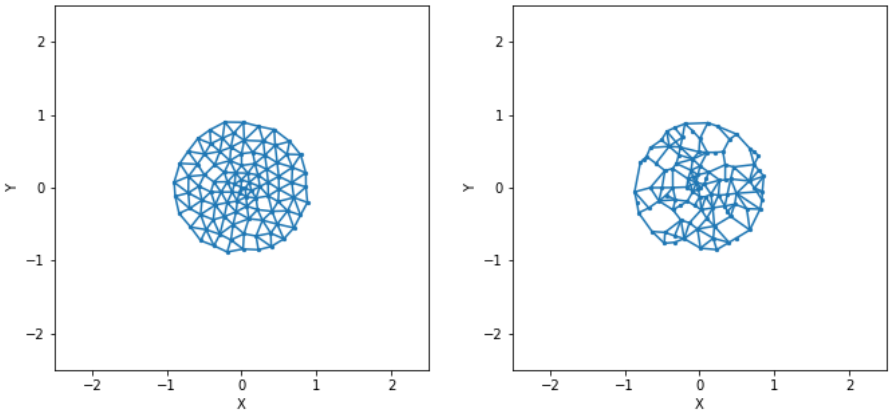
6.2 Anomaly detection

TiliNG was designed to cover all observations with minimal number of nodes, as evenly as possible. This means that even those samples that are rare and/or far from the majority of the samples will be represented. Nodes can exist even for the sake of one single observation. This behaviour might be desired to detect anomalies in the data. On Figure 26 we try to demonstrate this behaviour on a data-set, where we inserted random points to our data and then shuffled it.



(a) The pure data-points

(b) TiliNGs result



(c) Basic GNG's result

(d) GNG with Utility result

Figure 26: TiliNG compared to other GNGs (TiliNG made 102, the GNGs 100 neurons)

As it can be seen, the anomalies are clearly visible even after our algorithm finished. This can not be said in the case of the other GNGs. We designed TiliNG not to be affected by density, otherwise it would mean, that the algorithm would over-represent the dense areas and under-represent the anomalies, and we would need to produce considerably more BMUs to even display the anomalies. TiliNG treats the dense areas and the rare states the same way, thus avoiding the under-representation of anomalies and keeping the number of BMUs relatively low at the same time.

6.3 Wireless Sensor Network Topology and TiliNG

Typically sensor-networks are self organizing, the networks structures can not be planned, their physical topology can be considered random. We can however influence their logical topology, it is important to pay attention to scalability. The most common way to do this, is by dividing the network in clusters, such a way, that every node is part of at least one cluster and each cluster has a cluster-head, which controls the nodes in the cluster. Gateway stations ensure the communication between clusters. Now for each cluster to "know" every other cluster the cluster-heads are organised into a hierarchical tree and the resulting network is scalable and each node can communicate with any other node. It is however a hard problem (in fact it is considered NP-difficult) to find the optimal clustering of the nodes, although there are existing $O(n^2)$ complexity, but heuristic based algorithms to do the job. We believe that TiliNG might be able to help finding optimal clustering.

For this we need to think about our BMUs and their surrounding (circle with a radius R) area as the resulting clusters. Since TiliNG will cover every sample fed to it, with minimal overlap and therefore minimal number of BMUs, if we correspond the samples to the sensors locations (or additional attributes that can be interpreted as distances and that needs to be taken into consideration, while clustering), we end up with an optimal coverage (or at least an approximation of it). Note here that usually the sensors, after their deployment are stationary (depending on the application, they can be mobile, but even then they have low mobility), so their location can be handled as constants (as TiliNG assumes). The algorithm might also help building the cluster-heads tree, we just need to repeat it with a larger R on the cluster-head locations (or we can just use the BMUs location from the previous run, considering the cluster-heads are most likely going to be placed in their cluster).

References

[1] Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in neural information processing systems*, pages 625–632, 1995.

[2] Bernd Fritzke. A self-organizing network that can follow non-stationary distributions. In *International conference on artificial neural networks*, pages 613–618. Springer, 1997.

[3] Mohamed-Rafik Bouguelia, Yolande Belaïd, and Abdel Belaïd. An adaptive incremental clustering method based on the growing neural gas algorithm. In *2nd International Conference on Pattern Recognition Applications and Methods-ICPRAM 2013*, pages 42–49. SciTePress, 2013.

[4] H. E. Krogstad. *KARUSH-KUHN-TUCKER THEOREM*, 2012. https://folk.ntnu.no/hek/Optimering2012/kkttheoremv2012.pdf.

[5] Thomas Martinetz. Competitive hebbian learning rule forms perfectly topology preserving maps. In *International conference on artificial neural networks*, pages 427–434. Springer, 1993.

[6] *Boost C Libraries*. https://www.boost.org/.

[7] *OpenMP*, Nov 2018. https://www.openmp.org/.

[8] Hans-Peter Kriegel Stefan Berchtold, Daniel A. Keim. The x-tree: An index structure for high-dimensional data. *Communications in Statistics-theory and Methods*, 1996.

[9] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[10] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020.

[11] NetworkX developers. *Network Analysis in Python*, 2014. https://networkx.github.io/.

[12] Jiawei Han Xin Jin. *K Means Clustering*, 2011. https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-30164-8_425.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[14] Ricardo J. G. B. Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In Jian Pei, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu, editors, *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.