Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Prevention of occurrence of dead units in self-organizing maps

Master thesis

2018
Bc. Jakub Novák

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Prevention of occurrence of dead units in self-organizing maps

Master thesis

Study program:	Informatics
Field of study:	2508 Informatics
Department:	Department of Informatics
Supervisor:	doc. RNDr. Martin Takáč, PhD.

Bratislava, 2018
Bc. Jakub Novák

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Jakub Novák
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Prevention of occurrence of dead units in self-organizing maps

Annotation: One of the problems of self-organizing maps (SOM) is occurrence of so-called "dead units" that effectively lower the capacity of the SOM. Usually due to random weight initialization, some neurons have weights far from any data, hence never win the competition and rarely get any training that would pull them out of bad regions, hence they become "dead". The goal of this thesis is to review existing approaches to the problem, suggest several SOM modifications, implement simulations and analyse to what extent the proposed modifications succeed in eliminating dead units. This also includes exploring the space of free parameters of these modifications and finding the best values.

Literature: M. Van Hulle (2000): Self-organizing Maps. In: Handbook of Natural Computing, pp 585-622.
T. Villmann et al (2014): Advances in Self-Organizing Maps and Learning Vector Quantization. Proceedings of the 10th International Workshop WSOM.

Supervisor:	doc. RNDr. Martin Takáč, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	14.12.2016

Approved: 19.12.2016 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgement: First and foremost, I would like to thank my supervisor, Martin Takáč, for all the hours spent in conversation and endless e-mail chains about everything relevant to the topic of this thesis. I am very grateful to Marek Šuppa, Ondrej Jariabka and Peter Poláčik, for patiently answering all the technical and non-technical questions I have came up with over the course of finishing this thesis. Special thanks belongs to Diana Valková, for her endless moral support.

Abstrakt

Jeden z problémov samoorganizujúcich sa máp je výskyt mŕtvych neurónov, ktoré spôsobujú, že kapacita siete nie je plne využitá. Tento problém zväčša spôsobuje zlá náhodná inicializácia váh siete, ktorá spôsobí, že niektoré neuróny budú mať váhy ďaleko od ktoréhokoľvek vstupu. Toto spôsobí, že tieto neuróny nikdy nevyhrajú súťaž a s veľkou pravdepodobnosťou ani nebudú adaptované vítazovou susedskou funkciou.

V tejto práci navrhneme a otestujeme niekoľkých modifikácií učiaceho algoritmu samoorganizujúcich sa máp, ktoré by mali pomôcť pri prevencii mŕtvych neurónov, a zároveň preskúmame priestor parametrov navhrnutých metód a ich vplyv na výsledky.

Kľúčové slová: mŕtve neuróny, samoorganizácia, samoorganizujúce sa mapy, neurónové siete

Abstract

One of the problems of self-organizing maps is occurrence of so-called *dead units*, which cause the map to not be fully utilized. A bad initialization of weights can cause some neurons to have their weights far from any input data, rendering them useless, because they will never win the competition and, most likely, will not be in close proximity of winners neighborhood to be adapted by the neighborhood function.

In this thesis, we propose several modifications to self-organizing maps training algorithm aimed at preventing the occurrence of dead units, along with exploring their parameter space and how the parameter values influence the outcome.

Keywords: dead units, dead neurons, self-organization, self-organizing map, SOM, neural networks

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In this thesis, we will deal with self-organizing maps [3]. One of the main characteristics of self-organizing maps is that neurons remember information about available data in their weights. When the map is small, each neuron is forced to represent more different data, harming quality of clustering. When the map is bigger, neurons can represent the input data more precisely. However, just training a big self-organizing map is not enough, because unused neurons, called dead units, might occur.

In this thesis, we will experiment with different methods for preventing dead units in self-organizing maps. Dead units are not a very well explored field, therefore we will try several methods minimizing occurrence of dead units across various parameters.

This thesis is structured as follows: in the 1st chapter, we will formally and in detail describe self-organizing maps. In the 2nd chapter, we will show related work to the problem of dead units. In the 3rd chapter, we will describe methods we will be using to reduce the number of dead units in SOM. In the 4th chapter, we will show results of these methods and compare them. The final chapter will be the discussion.

1.2 Self-organizing map

A self-organizing map is a type of artificial neural network that is trained using unsupervised learning to produce a typically two-dimensional, representation of the input space, called a map. SOM is therefore a method of dimensionality reduction. The main difference between SOM and other artificial neural networks is that SOM utilizes competitive learning instead of error-correction learning (e.g. backpropagation with gradient descent). To preserve the topological properties of the input space, they use a neighborhood function [4].

Neuron arrangement represents input space in which the distance between two neurons is usually given by Euclidean distance of vectors of its coordinates. This topology preserving representation, created after SOM is fully trained, has important attribute: randomly chosen samples which are close to each other in input space also evoke responses on neurons that are physically close to each other on the map (output space).

1.2.1 Biological inspiration

Topological feature mapping has distinctive representation in biological neural networks—more specifically in brains of primates and humans. Topological maps introduce an effective way of representing important parameters of input data. It is experimentally proven that topological maps in brains are not fully developed after birth, but are formed during early stages of development. Further, it is proven that the process of modification takes place based on impulses coming from outside environments, therefore from the learning perspective, it is self-organized [5].

1.2.2 Structure of SOM

Self-organizing map consists of neurons, which create the map space. The map space is defined first and usually is a finite two-dimensional region with nodes arranged in a regular hexagonal or rectangular grid. Each node has its own weight vector, which represents a position in the input space; meaning it has the same dimension as each input vector. During training, neurons are fixed in the map space and their weight vectors are moved towards the input data, whilst topology stays preserved. SOM then describes a mapping from a higher-dimensional input space to a lower-dimensional map space. After training, SOM can be used to classify a vector from the input data by finding the winner neuron - neuron, whose weights vector is closest to the input datum.

1.2.3 Learning

The goal of learning in the SOM is to create similar responses to input patterns from each different part of the network. This is partly motivated by how visual, auditory or other sensory information is handled in separate parts of the cerebral cortex in the human brain [6].

There are two ways to initialize weights of neurons - either randomly to small values, or evenly from the subspace spanned by the two largest principal component eigenvectors of the dataset. When using the second way of initialization, learning becomes faster, because the initial weights of neurons are already good approximation

of SOM weights [7]. The network needs a large number of training input data to better approximate the vectors during the mapping. These are usually trained over several iterations.

The training utilizes competitive learning. Competitive learning is a form of unsupervised learning in artificial neural networks, where neurons compete between each other for the right to respond to the input datum. For each input datum fed to the network, Euclidean distance between this datum and all weights of neurons is computed. The neuron with weights closest, or the most similar, to the input datum is called the winner neuron, or the best matching unit (BMU). The weights of the winner neuron and weights of neurons within its neighborhood are then adapted towards the input vector. The amount of adaptation decreases with time and distance from the winner neuron.

$$i^* = arg\,min_i(d_E(\vec{x}, \vec{w}_i)) \tag{1.1}$$

$$d_E(\vec{x}, \vec{w}) = ||\vec{x} - \vec{w}|| \tag{1.2}$$

$$\vec{w}_i = \alpha \cdot (\vec{x} - \vec{w}_i) \cdot h(i, i^*) \tag{1.3}$$

$$h(i, i^*) = e^{-\frac{{d_E}^2(i^*, i)}{\lambda^2(t)}} \tag{1.4}$$

i^* is the winner neuron, d_E is the Euclidean distance between neuron i and input datum and α is learning rate. Neighborhood function always returns 1 for $i = i^*$. Equation 1.1 is used to find the winner neuron, 1.2 is used to calculate the Euclidean distance between input datum and neuron weights and 1.3 is used to adjust all neurons.

The neighborhood function $h(i, i^*)$ depends on the grid-distance between the BMU (neuron i^*) and neuron i. Most commonly, Gaussian function 1.4 is used to calculate neighborhood function. Neighborhood function shrinks with time - at the beginning of training, the neighborhood is big and self-organizing happens on a bigger scale. When the neighborhood is small enough to only affect some neurons, weights of neurons converge to local estimates. Usually, the learning rate α and the neighborhood function h decrease relative to number of current epoch.

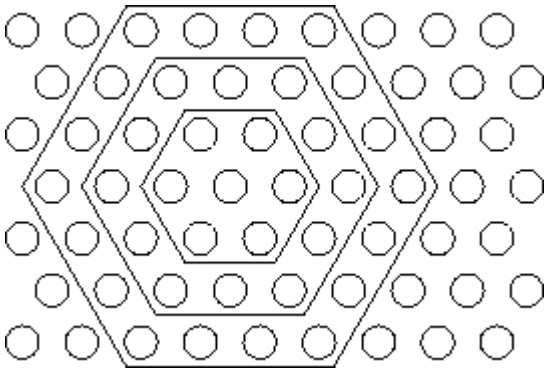


Figure 1.1: Neighborhood of a given winner unit[1].

This process repeats for each input datum for a number of epochs. The network ends up connecting output neurons with patterns from the input data.

During learning (or mapping), for each datum there is one winning neuron. This neurons weight vector is closest to the input vector. The distance between input datum and each neurons is calculated using the Euclidean distance.

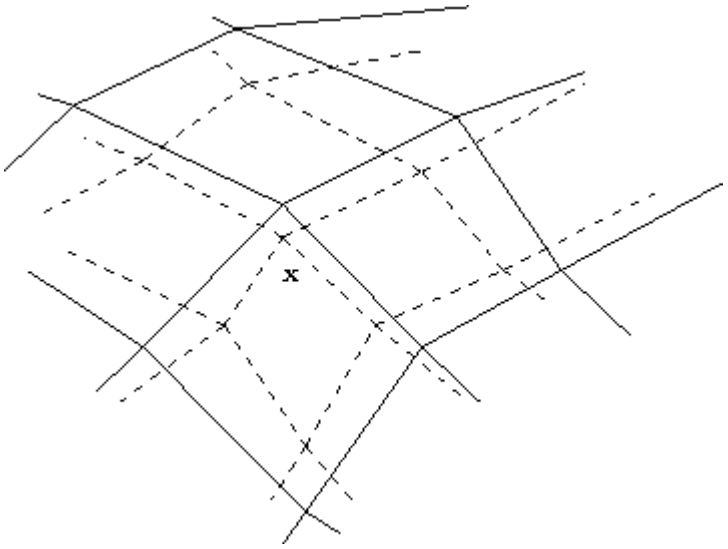


Figure 1.2: Updating the best matching unit and its neighbors[2].

[1]Taken from https://users.ics.aalto.fi/jhollmen/dippa/node9.html
[2]Taken from https://users.ics.aalto.fi/jhollmen/dippa/node19.html

Algorithm

1. Randomly initialize weights of neurons.
2. Select one data point (randomly or systematically cycle through dataset).
3. Find the neuron that is closest to the chosen data point (BMU).
4. Move the BMU closer to that data point.
5. Move the BMU's neighbor neurons closer to that data point with neighborhood function.
6. Update the learning rate and neighborhood function.
7. Iterate from step 2 until positions of neurons have been stabilized.

Initialization

Selection of a good initial approximation is a well-known problem for all iterative methods of learning neural networks. Kohonen used random initiation of SOM weights [8]. We will do the same.

1.3 Dead units

One of the problems of self-organizing maps is occurrence of "dead" units. Dead units are units whose randomly initialized weight vectors are so far from any data point, that they never get chosen as BMU, hence are never adapted to move closer to data (assuming that they are neither adapted because of being in the neighbourhood of another BMU). These dead units cause the network to not be fully utilized.

1.3.1 Causes

Dead neurons are usually caused by badly initialized weights in the SOM. Some neurons will have weights far from the input data, therefore they will not be adapted at all or will be adapted too little, if they are at the edge of the winner neurons neighborhood to be adapted by its neighborhood function.

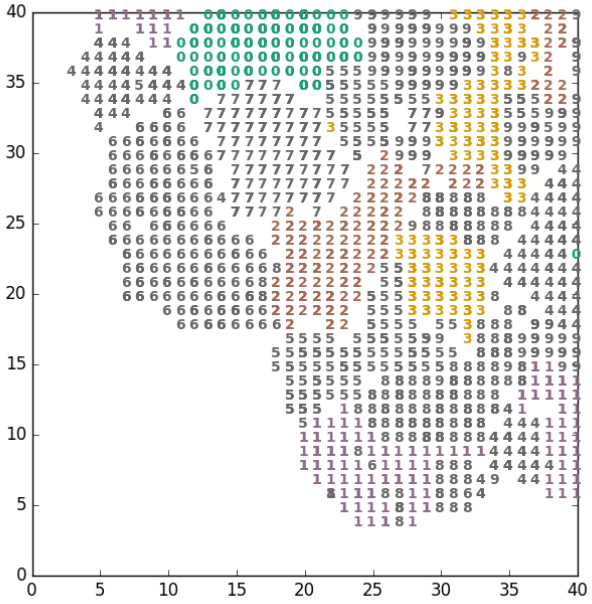


Figure 1.3: SOM (40x40) with dead units (38% of the map) trained on MNIST dataset.

Chapter 2

Related work

There are not many works focusing specifically on the problem of dead units. Most works mentioning dead units deal with them specifically for their own agenda and only briefly mention the method used. The following papers show methods that adjust parameters according to the input space and we can see that dead units are closely related to the neighborhood size and other SOM parameters.

2.1 PLSOM

In 2003 Berglund and Sitte proposed the parameters and update function should depend on maps conditions instead of external variables, like learning rate. As internal condition for scaling these variables, they selected the Euclidean distance from the current input to the closest weight vector, normalized (divided) by the maximum Euclidean distance from any input seen so far to its closest weight vector. If the normalized Euclidean distance is large, the map needs to change more to fit future inputs, and small, the fit is good already and map does not need to change much. The idea behind PLSOM is that parameters should not be determined by the number of epoch, but by how good the topological representation of the input state is [1].

2.1.1 Algorithm

ϵ is scaling variable depending on how good the fit of the weight vector of the winning neuron is to the last input.

$$\epsilon(t) = \frac{\|x(t) - w_c(t)\|^2}{\rho(t)}$$

$$\rho(t) = max(\|x(t) - w_c(t)\|^2, \rho(t-1)),$$

$$\rho(0) = \|x(0) - w_c(0)\|^2$$

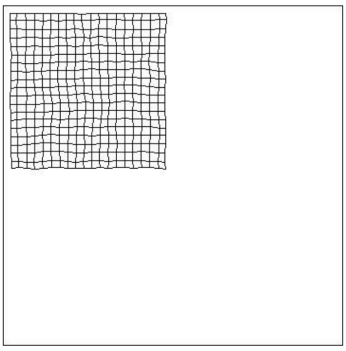
$\epsilon(t)$ represents Euclidean distance between the input datum and winner neuron at time t. If ϵ is too big, the map needs to readjust, since it is not fitting the data well and when the ϵ is small, input already fits the map well. In PLSOM algorithm, traditional annealing of the neighborhood is replaced with $\beta(t)$ =constant $\forall t$, because neighborhood size is determined by ϵ [1]. Equation for neighborhood function:

$$h_{ci}(t) = e^{\frac{-d(i,c)^2}{\Theta(t)^2}}$$

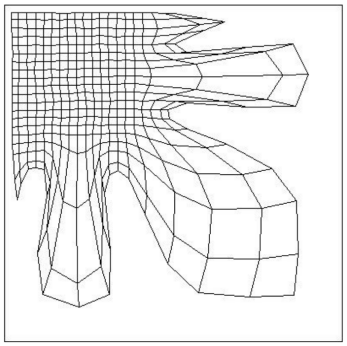
Equation for weight updates:

$$w_i(t+1) = w_i(t) + \epsilon(t)h_{ci}(t)[x(t) - w_i(t)]$$

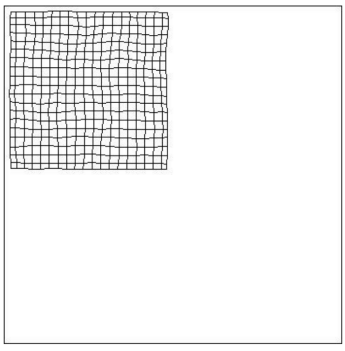
The learning rate has been eliminated and replaced by $\epsilon(t)$ In Figure 2.1 is the performance comparison of standard SOM algorithm, PLSOM and Matlab-implemented SOM algorithm from the paper.



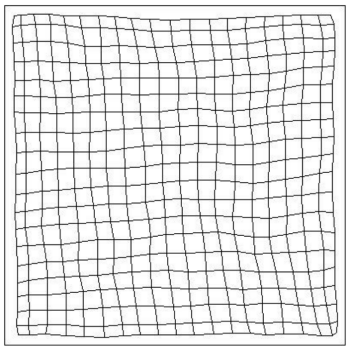
(a) SOM weight vector position in input space after training for 50000 iterations with uniformly distributed pseudo-random 2-dimensional input, ranging from 0 to 0.5.



(b) Same SOM as in figure (a) after 20000 further training steps with inputs ranging from 0 to 1.0.



(c) PLSOM weight vector position in input space after training for 50000 iterations with uniformly distributed pseudo-random 2-dimensional input, ranging from 0 to 0.5.



(d) Same PLSOM as in figure (c) after 20000 further training steps with inputs ranging from 0 to 1.0. Note the difference between this figure and figure (b).

Figure 2.1: Comparison of SOM and PLSOM [1].

2.2 PLSOM2

The earlier modification, PLSOM, solved some problems of SOM, but introduced other. One of them is overreaction to extreme outliers, which happens even when the SOM trains for a longer time period. This causes problems, since any standard, non-normalized dataset of acceptable size will most probably have outliers. Another problem PLSOM introduced is that initial weights can affect the amount of adaptation if the weights are initialized too far from the input space, causing training examples to cluster on the edges. PLSOM2, introduced in /cite, tries to fix these problems. PLSOM2, instead of using size of the error relative to the maximum error to change the scaling of the update, uses the range of observed inputs.

Even though the PLSOM was an overall improvement over standard SOM, the success of the SOM is heavily dependent on initial weights and the number of outliers in the dataset. Scaling of the update in PLSOM is based on $err(t)$ relative to the largest error seen up to that point. When weights are initialized far from input space, the initial error ($err(0)$) is very large and all further adaptations will be very small, causing the map to not be ordered at all and overreact to extreme outliers. PLSOM2 does not scale error as PLSOM does, instead it scales it relative to the diameter of the union of observed inputs /cite. This means that no activation map is stored, therefore the initial weight distribution does not get to influence the map after several epochs.

2.2.1 Algorithm

Algorithm performs two main steps for each input. The first step is determining the size of the input space (S), the second one is updating the weights. The weight update depends on the input space, since these operations run sequentially. The input space size is defined as the diameter of the dataset:

$$S_t = \max_{i,j}(\|x_i - x_j\|^2), \;\; i, j \leq t$$

where $x_i \in \mathbb{R}^n$ is the input at time i, t is the current time. The weight update is defined as:

$$d(t) = \min\left(\frac{err(t)}{S}, 1\right)$$

where $d(t)$ is scaling variable representing the fit and S is the input space. Neighborhood size is determined by $d(t)$ and

$$\Theta(d(t)) = \beta \ln(1 + d(t)(e - 1))$$

where $(e - 1)$ is the scaling factor, which is chosen to ensure the range of $d(t) = [0, 1]$ maps into range of $\Theta = [0, \beta]$ and β = constant $\forall t$ is neighborhood range. Neighborhood range does not change during training (unlike neighborhood size in SOM),

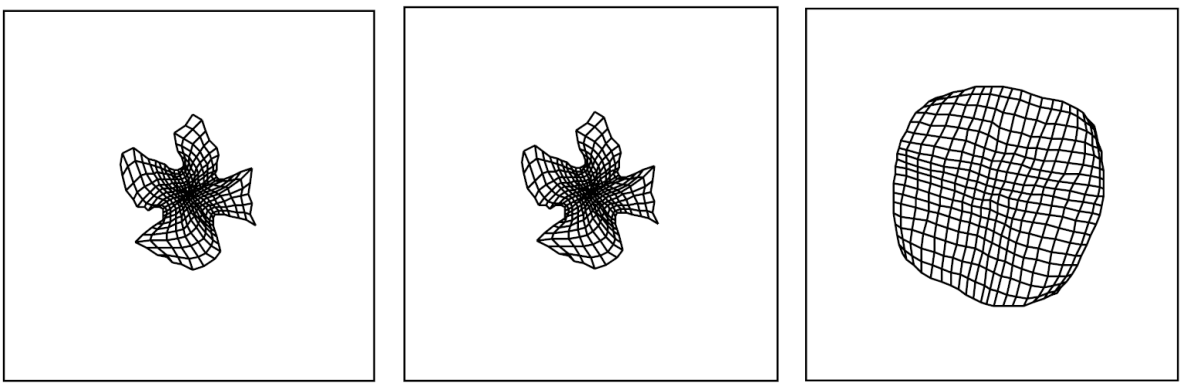
instead it is used indirectly in calculating the neighborhood size. The value of Θ is then used in the neighborhood function:

$$h_{c,i}(t) = e^{\frac{-D(i,c)^2}{\Theta(d(t))^2}}$$

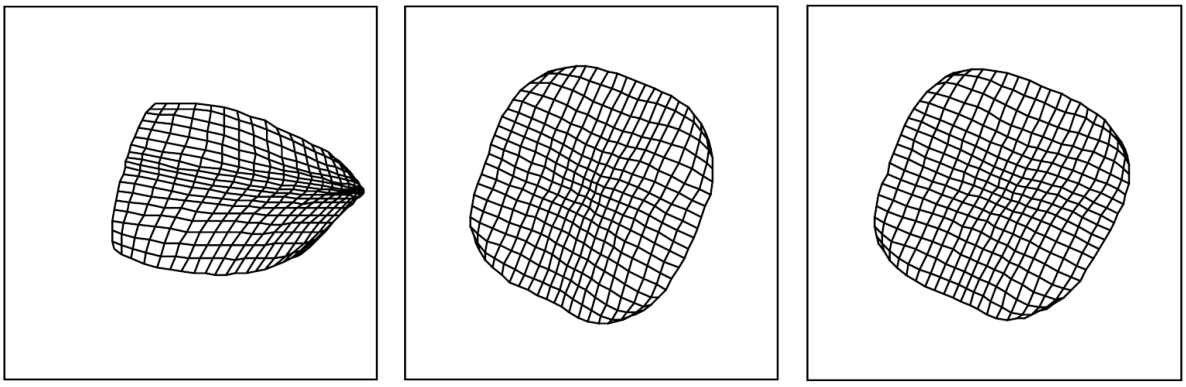
where $D(i, c)$ is a distance in output space from the winning node c to node i, which is the node currently being updated. This gives a value for $h_{c,i}(t)$ that decreases with increased distance from c, and the rate of decrease is determined by $d(t)$. Results from [2] use Euclidean distance for $D(i, c)$. The weight update functions are:

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

$$\Delta w_i(t) = d(t)h_{c,i}(t)[x(t) - w_i(t)]$$



(e) SOM before outlier. (f) SOM after outlier. (g) PLSOM before outlier.



(h) PLSOM after outlier. (i) PLSOM2 before outlier. (j) PLSOM2 after outlier.

Figure 2.2: Comparison of response to outlier of SOM, PLSOM and PLSOM2 from [2]

Chapter 3

Method

A standard SOM presents two types of problems: convergence and dead units. A well initialized SOM might already fit the data well, therefore saving computational time as a result. A bad initialization can cause some neurons to never be adapted or be adapted insignificantly, causing the map to not be fully utilized, therefore degrading the quality of data representation. These neurons are called dead units.

To get a better idea, how to reduce occurrence of dead units in self-organizing maps, we created a set of experiments.

3.1 General setup

We use self-organizing maps of size 20x20. As a training set we use the digits dataset [1] which is made up of 1797 hand written 8x8 images of digits. Input is then normalized by 3.1.

$$data = data/data.argmax() \tag{3.1}$$

We did some preliminary experiments in which the error does not drop much after 30 epochs, therefore we will be training for 30 epochs in all experiments. In each epoch, the SOM will train all 1797 data samples.

3.2 Neighborhood size and learning rate annealing

This method represents more of a standard approach to training self-organizing maps. Most methods utilize parameter annealing. The first phase of annealing, called **initial organization phase**, is to start off with a bigger neighborhood and quickly, within few epochs, get to a small neighborhood size. For neighborhood function, we use Gaussian

[1] http://scikit-learn.org/stable/auto_examples/datasets/plot_digits_last_image.html

neighborhood (Equation 1.4). The σ will be used in further text as a parameter representing neighborhood size. Bigger values of σ mean bigger neighborhood sizes of the winner neuron i^*. The second phase, called **fine-tuning phase**, is used for fine-tuning the neuron weights.

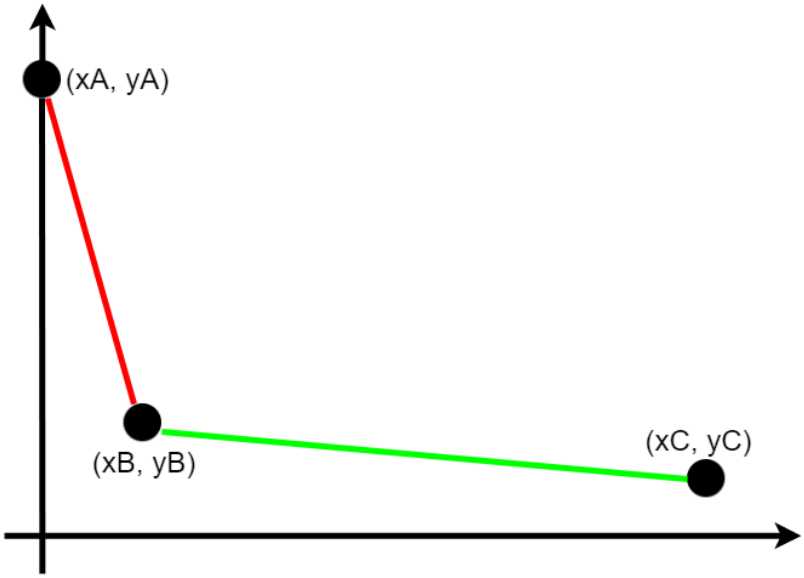


Figure 3.1: Annealing.

In Figure 3.1, the red line represents initial organization phase and the green line represents fine-tuning phase. xA is the first epoch and will always be 0, yA is the initial neighborhood size (and initial learning rate). xB is the number of epochs after which the breaking point occurs and yB is the neighborhood size (or learning rate) at this point. xC is always equal to maximum number of epochs. yC is the value of the neighborhood size (or learning rate) at the end of the training.

In our experiment, we fixed the following parameters for learning rate:

$$alpha\_yA = 1.0 \tag{3.2}$$

$$alpha\_yC = 0.1 \tag{3.3}$$

and for neighborhood size:

$$sigma\_yB = 1.0 \tag{3.4}$$

$$sigma\_yC = 0.5 \tag{3.5}$$

Both $alpha\_xA$, $sigma\_xA$ are 0 and $alpha\_xB$, $sigma\_xB$ are set to the breaking point.

Decay function used was:

$$decay = yA + \frac{(t - xA) * (yB - yA)}{(xB - xA)} \tag{3.6}$$

where t is the number of sample. The formula represents linear decay from (xA, yA) to (xB, yB).

We explored all combinations of breaking point learning rates {0.1, 0.2, 0.5}, initial neighborhood sizes {2.0, 5.0, 10.0, 20.0, 50.0, 100.0} and breaking points {1, 3, 7, 12, 20}. Our hypothesis is that starting with larger neighborhood sizes can help minimize the number of dead units. We also explore the parameter space of learning rate, because annealing the learning rate is the usual way of SOM training, but our main focus is on the influence of neighborhood size annealing scheme on the number of dead units.

3.3 Training random dead unit

In [9], the authors create an algorithm with two steps to prevent dead units occurrence. The algorithm starts off with a (usual) direct adaptation: the SOM is trained on a datum and the winning neuron and its neighborhood adapts. The proposed new step, inverted adaptation, randomly picks a dead neuron, finds the closest datum for the chosen neuron and adapts its weights towards the chosen datum. This creates a possibility that potential outliers that would otherwise never be adapted (not even by neighborhood function of other winner) will be adapted.

The authors, however, do not explore how much the method affects the occurrence of dead units; they rather focus on using this method to represent a 3D space in 2D. This is why we decided to re-implement this method, explore it deeper and compare to other methods.

Our method is a modified version of the method proposed in [9]. In standard training steps, we train the SOM on the dataset, whilst keeping a map of all neurons that have not been trained yet. After each standard step, we randomly choose a dead unit, find the closest input datum and then adapt its weights and weights of neurons within the winner neurons neighborhood with this datum.

We tested two variations of this method. One, where we remove the dead unit from the list of dead units after it gets trained and the other, where we do not remove the dead unit from the list.

3.4 Training dead units for novel inputs

The basic idea of the third method is that if the winner neuron has its weights similar enough to the input datum, the input is most probably an instance of the same category and the winner can be adapted. If not, rather than overwriting the winner, it is better to choose a different unused neuron and adapt that one (and its neighborhood) instead.

To do so, we find the closest dead unit to the datum and adapt it.

For this method, we needed to determine a threshold on the input–winner distance that would distinguish digits within the same category from digits across categories. We quickly faced the problem of data separability, which is the property of the data we are using to train the SOM. Reasonable threshold values to experiment with turned out to be within the approximate range 0.4-0.6, so we chose 0.4, 0.5, and 0.6 (more on how we calculated these thresholds in Section 4.3).

We decided to experiment with two variants of this method. In the first variant, dead units are adapted with a learning rate according to the annealing scheme. In the second variant, dead units are adapted with a constant learning rate 1.0.

3.5 Implementation

3.5.1 Programming language

We used Python as our primary programming language. We chose this language because of its ease of use and the large variety of libraries supporting machine learning and neural networks training.

Python

Python is dynamic, high-level programming language. It is meant to be an easily readable language, which uses whitespace indentation, rather than curly brackets or keywords, to delimit blocks.

Python is an open-source project and its interpreters are available for many operating systems. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

3.5.2 Libraries

We used variety of libraries. The most important are listed below.

NumPy

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

NumPy targets the CPython reference implementation of Python, which is a non-optimizing bytecode interpreter. Mathematical algorithms written for this version of Python often run much slower than compiled equivalents. NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays

Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

scikit-learn

Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Chapter 4

Experiments

4.1 Neighbourhood size and learning rate annealing

A well selected scheme for neighborhood size and learning rate annealing as well as the right combination of parameters can significantly influence the outcome of training. Therefore we decided to explore the parameter space, explained in more detail in Section 3.2.

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	13.50%	11.50%	10.50%	11.75%	12.25%	10.50%
	3	12.25%	10.75%	12.25%	12.75%	11.50%	11.75%
	7	10.30%	11.30%	9.50%	10.00%	11.50%	12.80%
	12	8.75%	8.25%	11.75%	9.75%	10.00%	10.25%
	20	11.25%	10.50%	11.00%	10.50%	11.25%	11.25%

(a) 0.1 learning rate at breaking point

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	13.00%	11.25%	9.50%	12.75%	12.00%	11.75%
	3	10.50%	9.00%	13.50%	7.50%	13.25%	11.75%
	7	10.50%	13.00%	7.00%	10.50%	11.30%	10.50%
	12	12.25%	8.75%	9.50%	11.25%	7.75%	9.00%
	20	10.00%	10.50%	9.75%	10.00%	10.75%	9.75%

(b) 0.2 learning rate at breaking point

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	8.75%	9.50%	9.75%	11.25%	9.75%	9.25%
	3	10.25%	8.50%	11.00%	9.25%	11.00%	11.25%
	7	11.00%	12.00%	11.50%	12.50%	11.50%	10.50%
	12	10.00%	8.50%	9.50%	10.00%	10.75%	10.00%
	20	11.50%	12.00%	11.25%	9.75%	12.25%	12.50%

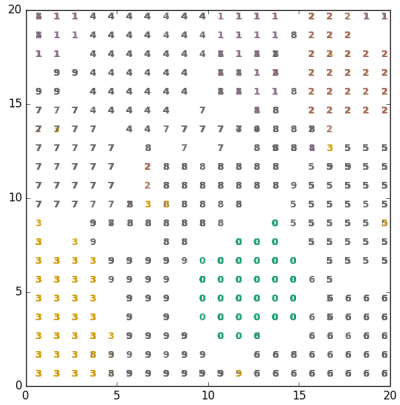
(c) 0.5 learning rate at breaking point

Figure 4.1: Results for parameter combinations on standard SOM.

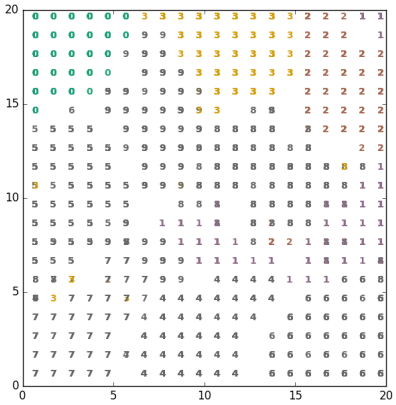
All schemes and parameter combinations resulted in 7-13% dead units after 30 epochs, with the best results having 0.2 learning rate at breaking point. We had expected a bigger initial sigma to have better results, but we disproved this hypothesis. Initial sigma of 50.0 and 100.0 for 20x20 SOM seems to be, on average, too big.

We can observe an interaction between initial sigma and breaking point approximately in the form of a diagonal (from bottom left to top right). This means the bigger

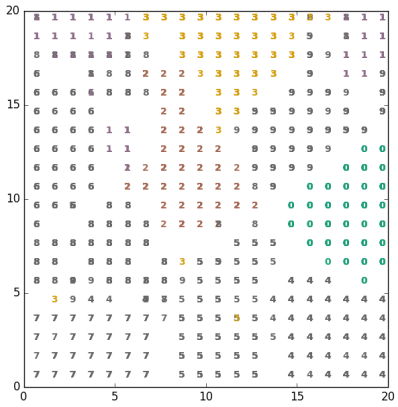
sigma, the sooner the breaking point should be. We think it is because when sigma is too big, it overrides a big part of the map, which is good, but not for too many epochs.



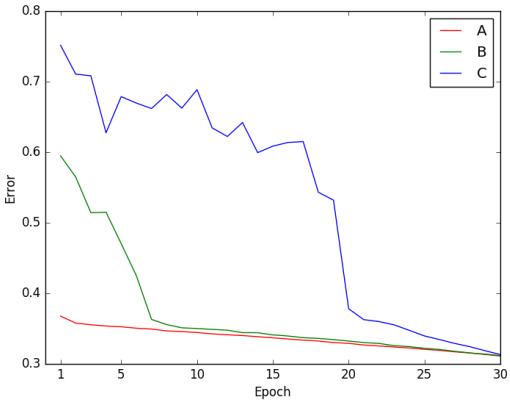
(a) The worst result for 0.1 breaking point alpha (13.5% dead units)



(b) Best result (7% of dead units)



(c) The worst result for 0.5 breaking point alpha (12.5% dead units)



(d) Quantization error

Figure 4.2: Winmaps and quantization error for the best and the worst combinations of parameters.

As we can see in Figure 4.2a-c, all three runs ended up with a good topographical organization, so in this sense there are no important differences. When we look at the quantization error graph (Figure 4.2d) we can see that all the methods ended up with approximately the same error, however there is a difference in how fast the quantization error decays.

As we can further see in Figure 4.2d, "breaking points" of the curves in which the error drops coincide with the breaking points of the used annealing schema: for the red line it is 1 (its sigma is 1 after the first epoch). The green line gets to that same

point after the 7th epoch and the blue line after the 20th epoch. We conclude that, no matter how many dead units there are, the error is big as long as the sigma is big. This is most likely happening, because different input data keep overwriting a part of the map representing other data.

4.2 Train random dead unit

In this method, after a normal training step, we choose a random dead unit from the set of dead units, find the closest datum to this neuron and adapt its weights towards this datum.

We decided to execute two versions of this experiment. The first version keeps the updated unit in the set of dead units, the second version removes it.

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	13.75%	12.00%	12.75%	12.50%	12.50%	10.00%
	3	10.00%	10.75%	12.25%	9.25%	12.50%	10.75%
	7	11.25%	10.25%	9.25%	10.50%	12.75%	11.50%
	12	10.75%	11.25%	8.25%	9.25%	12.25%	11.50%
	20	8.25%	10.50%	12.25%	10.50%	10.00%	11.50%

(a) 0.1 learning rate at breaking point

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	11.75%	11.50%	12.25%	10.75%	11.00%	10.75%
	3	10.50%	7.50%	9.00%	10.50%	10.50%	12.50%
	7	10.00%	10.75%	9.75%	10.50%	12.75%	11.75%
	12	9.75%	10.50%	10.75%	9.75%	10.50%	12.50%
	20	10.00%	9.25%	10.50%	12.50%	13.50%	11.50%

(b) 0.1 learning rate at breaking point

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	11.75%	10.25%	11.50%	10.50%	11.00%	11.50%
	3	10.25%	11.00%	12.25%	9.50%	11.75%	11.25%
	7	11.00%	9.00%	11.50%	10.75%	10.25%	11.50%
	12	9.00%	10.00%	9.00%	10.00%	9.25%	9.25%
	20	11.75%	11.00%	10.25%	9.75%	12.50%	10.25%

(c) 0.2 learning rate at breaking point

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	11.75%	9.75%	13.00%	8.00%	11.00%	10.50%
	3	12.75%	11.50%	11.50%	11.25%	12.25%	10.00%
	7	13.00%	10.25%	10.00%	13.00%	13.25%	12.50%
	12	7.00%	11.25%	11.00%	10.00%	12.00%	9.25%
	20	11.00%	12.50%	11.25%	11.00%	9.50%	12.50%

(d) 0.2 learning rate at breaking point

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	9.50%	10.25%	0.115	11.25%	10.75%	10.25%
	3	10.25%	11.00%	10.50%	11.75%	12.50%	9.50%
	7	10.25%	11.75%	12.00%	11.00%	10.00%	11.50%
	12	11.75%	10.75%	9.50%	10.75%	10.25%	10.75%
	20	11.00%	10.75%	9.00%	13.75%	11.00%	11.50%

(e) 0.5 learning rate at breaking point

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	10.50%	12.00%	9.00%	11.75%	8.75%	10.25%
	3	12.25%	11.25%	10.25%	12.25%	13.00%	9.50%
	7	10.75%	14.00%	10.75%	9.50%	11.00%	10.75%
	12	8.25%	10.50%	11.75%	10.25%	12.25%	10.75%
	20	10.75%	10.00%	11.00%	11.00%	13.50%	11.00%

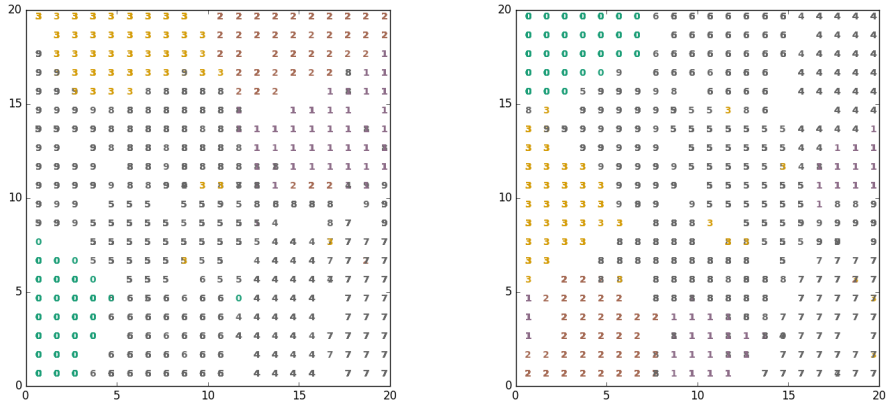
(f) 0.5 learning rate at breaking point

Figure 4.3: Results for parameter combinations. On the left side, removing the dead unit from set of dead units technique was used. On the right side, keeping the dead unit in the dead units set technique was used.

The results, again, show no significant differences. All parameter combinations resulted in 8.25-13.75% dead units when the dead unit was removed from the set of dead units after its adaptation and 7-13% when the dead unit was kept in the set. By

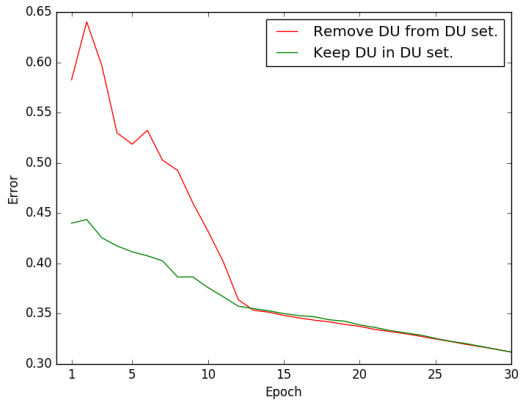
visual inspection of the tables it seems that keeping the dead units in the set achieved slightly better results.

We observe that the best results come from parameter combinations in the middle of the tables (see Figure 4.3), although it is not as consistent as for the first method.



(a) Remove DU from DU set (8.25% dead units)

(b) Keep DU in DU set (7% dead units)



(c) Quantization error

Figure 4.4: Comparison of the best results from two sub-experiments of train random dead unit experiment.

In Figure 4.4c we can see, that in terms of quantization error, both variants of the method end up with a similar error, but the one that keeps the trained dead units in the dead units set stabilizes the map more quickly.

However, this may have been caused by different initial sigmas in the two runs: the variant with removing DU had the initial sigma of 10.0, while the other one had the initial sigma of 2.0. The graph shows a similar trend to that on Figure 4.2d, so, by the same reasoning, when the training starts with sigma = 10.0, it takes sigma a longer

time to drop reasonably small values than in the run starting with sigma = 2.0, hence the error drops more slowly.

In comparison with the baseline experiment (neighborhood size annealing scheme), training dead units does not seem to be a significant improvement, both in terms of number of dead units and the quantization error.

4.3 Train dead units for novel inputs

In this method, during training we check whether the Euclidean distance of the winner neuron is greater or equal to a preset threshold (if not, it suggests novelty and a dead unit is trained instead of the found winner). We tested two variants: The first one trained the chosen dead unit with the learning rate according to the annealing scheme, the second one with learning rate 1.0. In our hypothesis, we wanted to find out whether it is effective to temporarily increase the learning rate to increase the learning capabilities of the chosen dead unit in case of a novel input (one-shot learning).

This method has proven to be quite sensitive to the novelty threshold, which should be set to a value that can distinguish digits within the same category from digits across categories.

Thus we experimentally measured Euclidean distances within and across some categories—we chose digits 3 and 8 for these estimations, since these two numbers look similar enough to confuse the network. Mean value of Euclidean distances between digits 3 was 0.45, between digits 8 was 0.49. Mean value of Euclidean distances between digits 3 and 8 was 0.58.

Reasonable threshold values to experiment with turned out to be within the approximate range 0.4-0.6, so we chose 0.4, 0.5, and 0.6.

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	11.00%	11.50%	8.50%	10.00%	10.50%	11.50%
	3	13.25%	13.00%	10.50%	10.50%	10.75%	9.25%
	7	10.00%	7.50%	10.50%	10.50%	9.75%	11.00%
	12	13.25%	9.50%	12.25%	8.00%	11.50%	10.75%
	20	10.50%	12.75%	9.00%	12.25%	12.25%	12.75%

(a) Threshold set to 0.4

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	13.80%	13.30%	13.50%	14.30%	15.00%	9.80%
	3	11.50%	12.00%	13.30%	15.30%	12.30%	12.50%
	7	10.80%	12.30%	12.00%	14.00%	10.00%	13.80%
	12	10.30%	11.30%	10.00%	10.80%	13.50%	15.50%
	20	11.30%	9.00%	9.80%	14.00%	13.30%	15.00%

(b) Threshold set to 0.4

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	12.00%	11.75%	11.25%	11.00%	12.25%	11.25%
	3	11.75%	12.00%	9.25%	9.75%	10.50%	8.25%
	7	12.00%	9.25%	9.75%	11.25%	12.25%	11.75%
	12	9.50%	8.75%	13.50%	10.00%	11.50%	12.25%
	20	12.00%	10.50%	9.50%	11.25%	11.25%	12.75%

(c) Threshold set to 0.5

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	45.00%	50.80%	15.50%	43.80%	41.50%	46.50%
	3	13.30%	42.00%	45.30%	46.30%	45.00%	39.30%
	7	11.50%	10.30%	12.30%	43.50%	47.30%	18.00%
	12	12.30%	14.00%	11.30%	15.50%	50.00%	43.50%
	20	13.50%	12.80%	14.80%	12.30%	15.00%	48.50%

(d) Threshold set to 0.5

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	12.75%	11.50%	9.75%	11.75%	13.50%	11.50%
	3	13.25%	13.50%	10.75%	12.00%	10.50%	10.00%
	7	9.00%	10.25%	9.50%	11.00%	11.75%	12.50%
	12	9.25%	10.25%	11.25%	10.00%	10.00%	11.50%
	20	9.25%	12.25%	10.50%	12.25%	12.25%	12.25%

(e) Threshold set to 0.6

Breaking point alpha: 0.1		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	23.80%	25.00%	27.00%	22.80%	24.80%	26.80%
	3	32.80%	32.30%	34.30%	26.50%	25.30%	26.00%
	7	41.80%	35.50%	58.00%	25.80%	25.00%	26.80%
	12	46.00%	12.30%	15.00%	48.80%	31.50%	31.30%
	20	37.00%	23.50%	29.00%	65.00%	42.00%	41.50%

(f) Threshold set to 0.6

Figure 4.5: Results for parameter combinations for 0.1 learning rate at breaking point. On the left side, decayed learning rate to update dead unit was used. On the right side, dead unit was updated using 1.0 learning rate.

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	12.00%	10.75%	11.50%	11.00%	9.25%	13.00%
	3	8.75%	11.50%	11.50%	10.00%	12.00%	10.25%
	7	11.75%	12.25%	12.50%	8.75%	11.50%	11.75%
	12	10.25%	10.25%	11.75%	12.00%	9.00%	11.00%
	20	10.50%	12.25%	12.00%	12.25%	12.00%	9.00%

(a) Threshold set to 0.4

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	10.80%	16.50%	14.00%	13.50%	13.30%	13.00%
	3	10.30%	11.00%	11.50%	13.50%	13.50%	11.50%
	7	10.00%	12.00%	11.50%	12.30%	10.80%	14.30%
	12	9.30%	10.80%	9.80%	11.80%	11.00%	14.00%
	20	9.50%	10.00%	11.00%	10.80%	13.00%	14.00%

(b) Threshold set to 0.4

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	10.25%	12.75%	12.50%	12.50%	13.25%	11.25%
	3	8.50%	10.75%	10.50%	10.00%	9.50%	10.00%
	7	11.50%	10.25%	10.50%	11.25%	9.25%	10.75%
	12	10.25%	11.50%	9.50%	11.25%	11.50%	11.25%
	20	11.00%	10.25%	12.25%	11.00%	12.00%	12.75%

(c) Threshold set to 0.5

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	43.50%	44.50%	42.80%	49.30%	45.80%	42.30%
	3	13.30%	40.00%	45.80%	37.30%	46.00%	42.80%
	7	10.00%	9.30%	8.80%	37.50%	43.00%	44.80%
	12	11.80%	13.00%	9.50%	12.30%	39.80%	42.30%
	20	13.30%	8.80%	13.30%	12.50%	33.80%	51.30%

(d) Threshold set to 0.5

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	13.50%	11.50%	11.25%	10.50%	13.00%	11.00%
	3	11.00%	12.25%	11.00%	10.75%	12.50%	10.00%
	7	8.75%	12.00%	9.25%	11.25%	12.00%	11.75%
	12	12.25%	11.25%	10.75%	12.25%	13.00%	11.50%
	20	10.00%	10.50%	9.75%	8.50%	12.25%	8.75%

(e) Threshold set to 0.6

Breaking point alpha: 0.2		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	31.30%	26.50%	26.50%	26.30%	25.00%	25.00%
	3	36.00%	35.50%	29.80%	25.80%	25.80%	20.30%
	7	39.30%	11.80%	14.00%	34.80%	26.30%	26.30%
	12	36.00%	15.00%	12.80%	47.50%	30.50%	26.50%
	20	27.30%	16.00%	35.00%	63.30%	54.30%	43.50%

(f) Threshold set to 0.6

Figure 4.6: Results for parameter combinations for 0.2 learning rate at breaking point. On the left side, decayed learning rate to update dead unit was used. On the right side, dead unit was updated using 1.0 learning rate.

The comparison between using annealed learning rate versus constant 1.0 for novel inputs shows that the constant 1.0 gives much worse results in terms of dead units (the right column in Figures 4.5, 4.6, and 4.7), where in extreme case it reaches values over 60% (except for novelty threshold 0.4 where the results are only slightly worse or comparable). As for novelty threshold, the values of 0.4 and 0.5 are slightly better than 0.6. As for the influence of learning rate value in the breaking point, the differences are minimal. The variability of results is very high for bad combinations (learning rate 1.0 for novel input, novelty threshold 0.5 and 0.6), where the range is 8.8-65%. For good combinations (the rest) the range is 7.5-13%.

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	11.50%	9.50%	8.75%	9.00%	13.25%	11.25%
	3	12.00%	9.00%	10.75%	12.75%	13.50%	10.75%
	7	11.50%	9.00%	13.50%	10.75%	12.75%	10.00%
	12	8.25%	11.50%	10.00%	11.00%	10.25%	10.00%
	20	11.75%	9.25%	12.00%	10.50%	10.50%	12.00%

(a) Threshold set to 0.4

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	10.80%	11.30%	11.80%	9.30%	11.00%	9.80%
	3	10.00%	10.50%	12.00%	10.30%	11.50%	11.80%
	7	12.30%	9.00%	10.30%	13.80%	10.50%	11.00%
	12	12.30%	9.80%	11.50%	10.30%	9.30%	13.30%
	20	12.50%	10.00%	9.80%	9.30%	12.30%	12.50%

(b) Threshold set to 0.4

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	11.50%	11.25%	11.75%	13.25%	9.50%	12.00%
	3	10.75%	12.25%	11.75%	10.00%	12.50%	8.50%
	7	9.25%	12.50%	9.50%	10.75%	11.50%	11.00%
	12	10.50%	10.75%	10.75%	10.75%	7.50%	10.25%
	20	12.50%	9.00%	13.00%	9.75%	11.00%	11.25%

(c) Threshold set to 0.5

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	33.80%	41.00%	36.50%	16.00%	16.80%	34.30%
	3	11.50%	38.00%	16.30%	33.00%	34.50%	33.00%
	7	10.80%	12.50%	11.80%	18.50%	17.80%	27.30%
	12	11.30%	10.30%	11.80%	12.00%	40.30%	38.00%
	20	13.00%	10.80%	11.80%	12.80%	39.50%	35.80%

(d) Threshold set to 0.5

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	12.00%	9.50%	11.00%	10.00%	11.00%	10.25%
	3	9.00%	11.50%	10.25%	12.00%	11.00%	12.25%
	7	11.50%	12.25%	11.50%	9.75%	12.50%	11.00%
	12	9.00%	11.00%	11.50%	12.25%	9.75%	10.25%
	20	8.75%	11.00%	11.00%	12.25%	10.75%	11.00%

(e) Threshold set to 0.6

Breaking point alpha: 0.5		Initial sigma					
		2.0	5.0	10.0	20.0	50.0	100.0
Breaking point	1	25.80%	26.30%	22.80%	22.00%	26.00%	22.50%
	3	31.80%	33.80%	40.00%	25.30%	28.30%	21.50%
	7	34.00%	11.30%	46.50%	32.80%	25.30%	31.30%
	12	35.00%	15.30%	13.30%	37.80%	29.30%	18.80%
	20	34.00%	23.30%	18.80%	61.50%	50.30%	53.30%

(f) Threshold set to 0.6

Figure 4.7: Results for parameter combinations for 0.5 learning rate at breaking point. On the left side, decayed learning rate to update dead unit was used. On the right side, dead unit was updated using 1.0 learning rate.

In terms of dead units, the best version of this method with novel input learning rate adjusting according to annealing scheme (threshold 0.4 and learning rate in the breaking point 0.1, or threshold 0.5 and learning rate 0.5) is comparable to the other two methods (Sections 4.1 and 4.2). We will return to across-method comparison in the next section.

4.4 Across method comparison

Because of a great number of parameter combinations, in all previous simulations we only did one run per combination to get an approximate idea about the parameter space. For a direct comparison of the three methods, we chose the best combination of parameters for each of them, run it five times with different random weight initialization and average the results to eliminate the influence of random factors.

We explored the following parameter combinations that gave the best results in pretests:

- Neighbourhood size and learning rate annealing
  - Initial sigma = 10.0
  - Breaking point = 7
  - Learning rate at breaking point = 0.2
- Train random dead unit
  - Variant: Keeping dead unit in dead units set
  - Initial sigma = 2.0
  - Breaking point = 12
  - Learning rate at breaking point = 0.2
- Train dead units for novel inputs
  - Threshold = 0.5
  - Initial sigma = 50.0
  - Breaking point = 12
  - Learning rate at breaking point = 0.5

Figure 4.8: Error and standard deviation across all methods. Each bar represents the average of 5 runs with different random initial weights of each method trained with the best parameter combination.

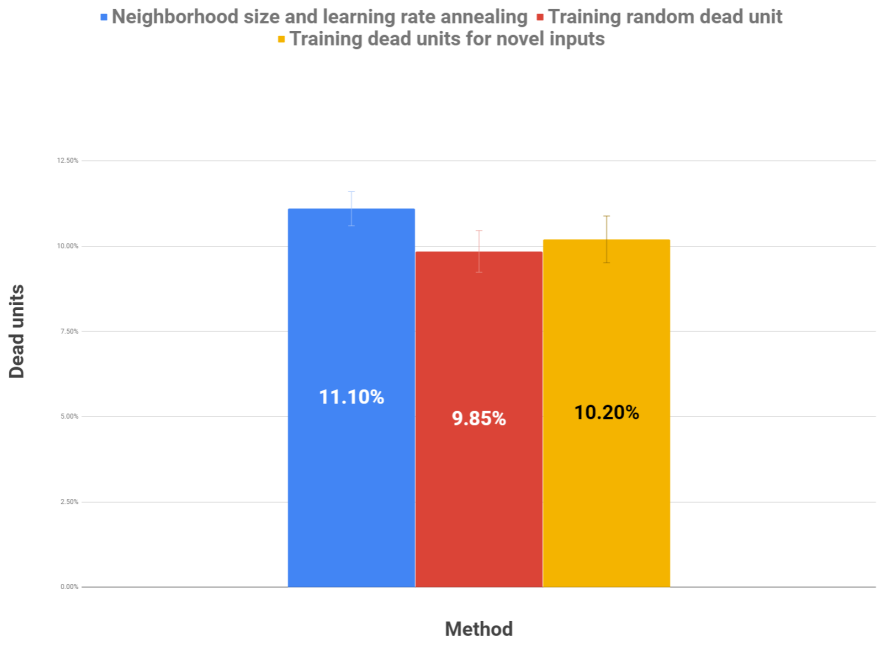


Figure 4.9: Number of dead units in final SOM across all methods. Each bar represents the average of 5 runs with different random initial weights of each method trained with the best parameter combination.

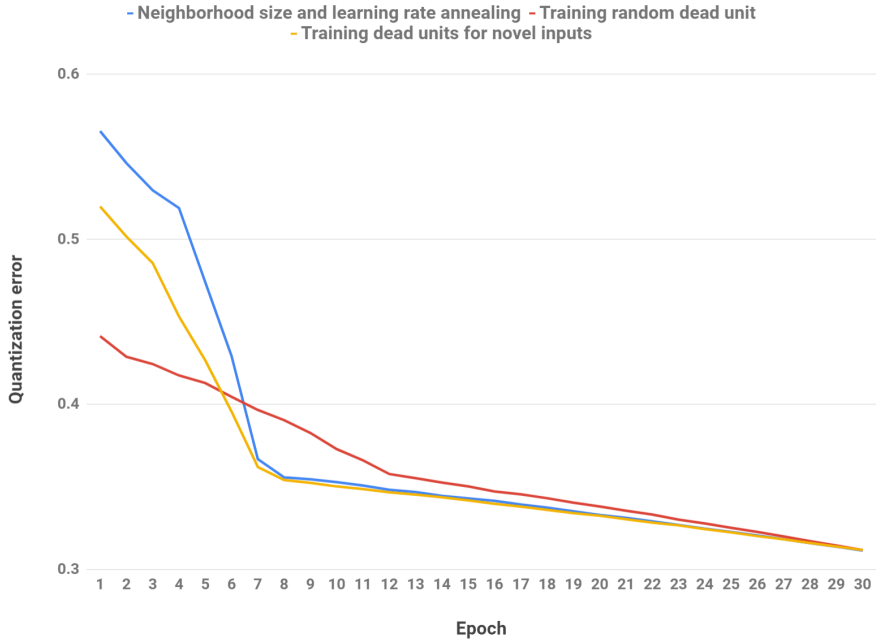


Figure 4.10: Quantization error during training.

We can see in Figure 4.8 that there are minimal differences relative to big variance overlap, therefore there is no clear winner method. This is also true for quantization error and dead units.

Finally, Figure 4.10 shows the quantization error during training. Although all the methods finally converge to similar value, the first method seem to be slower.

4.4.1 Complexity analysis

SOM has three input parameters: SOM size N, input dimension d and training set size T. Which for our case is: $N = 20\text{x}20 = 400$, $d = 8\text{x}8 = 64$ and $T = 1797$.

To update weights, we need $N * d$ space and $O(d)$ for Euclidean distance and $O(N)$ to find the winner, by comparing the input datum with weights of each neuron. This gives us time complexity $O(N * d * T)$ for each epoch.

In addition to the standard training step, the second method also needs to choose a random dead unit ($O(1)$) and compare it to each datum in the training set ($O(T)$), times $O(d)$ for each Euclidean distance calculated. This method then gives us $O(T * (d * N + d * T))$ instead of $O(T * d * N)$ in the first method. This is still linear in d and N, but quadratic in T. Therefore, before using this method, it should be considered on what data it is going to be used. If there is going to be a lot of data (T), it might not be the wisest choice, but it should run well on big vectors (d), e.g. camera images, or self-organizing maps with big sizes (N).

The third method compares the Euclidean distance between the input and the found winner to a threshold in each epoch and, in case of novelty input, it compares this input datum to each dead unit from set of dead units. The size of the dead units set can be N at most. This gives us (in addition to $O(N * d)$ for the standard case) another at most $O(N * d)$ operations in case of novelty. Thus the third method still runs in $O(T * N * d)$ time.

This means the second method is slower than both first and third method and should be used with caution in case of large data sets.

Chapter 5

Discussion

The goal of this thesis was to create and experiment with different methods for preventing dead units in self-organizing maps. We explored several methods across many parameters.

First was **Neighborhood size and learning rate annealing**, which is more of a standard approach to training SOM and we used it as a baseline.

Second was **Training random dead unit**, where after each standard step, we randomly chose a dead unit, found the closest input datum and then adapted its weights and weights of neurons withing the winner neurons neighborhood with this datum. We tested two variants of this approach - in the first variant, we removed the dead unit from the list of dead units after it got trained, and in the second variant, we did not remove the dead unit from the list.

The third method, **Training dead units for novel inputs** we tested, whether neurons weights were similar enough to the input datum, for which created a threshold variable. We had hoped, this approach would let inputs of the same category to be clustered better together and novelty inputs would be mapped to unused neurons. Since we introduced a new threshold variable, we had to determine, what would be the correct value to distinguish inputs within the same category from the ones across categories. We calculated the mean Euclidean distance for inputs of the same category as well as mean of cross-category Euclidean distances. We also decided to experiment with two variances of this method. The first variant adapted dead units with a learning rate according to the annealing scheme. The second variant adapted dead units with a constant learning rate of 1.0.

5.1 Results

From quantitative point of view, there were no significant differences in the final count of dead units, nor quantization error across methods except for the rate at which the quantization error converged. That is it why we will compare the methods based on what we believe they need to improve them.

In the first method, suitable values of initial sigma might depend on the size of SOM (the bigger map, the bigger initial sigma). Because of this, second methods looks the most promising.

The third method in its current state does not seem to be very reliable, since it requires to experimentally find out the correct threshold, which is extremely data-dependent.

For the third method, we have come up with another experiment, but time did not allow us to test it properly. The idea behind it was to remember, for each neuron, how much training it got. If the winner neuron did not satisfy threshold, we would choose a neuron with the least training received. Early results of this method have shown that this approach might not be very good for self-organizing maps as the under-trained neurons seemed to be disrupting clustering property of SOM.

5.1.1 Limitations

After seeing the final results, we consider the biggest limitation of this thesis was not trying to explore different SOM sizes, in addition to other parameters we explored and using only one dataset to test our hypotheses. The main reason was the computational and time demands of running that many different combinations of parameters.

In the future, we would definitely like to explore more SOM sizes, which have more neurons than training examples and see whether the bigger map gets fully utilized with quantization error dropping below current 0.31 (ideally close to 0).

Since all methods we tried ended up with very similar results, we would like to find a problem where differences would be more prominent.

In the future, a finer exploration of space of threshold values in 3$^{\text{rd}}$ might help this method to be substantially better. Also, seeing limitations of the approaches we used for this method, making sure the choice of dead unit does not disrupt the existing topographic organization will be quite important. An improvement in this direction can consist in choosing a unit to be trained on novel inputs based on a criterion function combining the similarity between the weights and input with the amount of training the unit received.

Bibliography

[1] E. Berglund and J. Sitte, "The parameter-less SOM algorithm," in *Proc. ANZIIS*, pp. 159–164, 2003.

[2] E. Berglund, "Improved PLSOM algorithm," *Applied Intelligence*, vol. 32, no. 1, pp. 122–130, 2010.

[3] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological cybernetics*, vol. 43, no. 1, pp. 59–69, 1982.

[4] T. Kohonen, "The self-organizing map," *Neurocomputing*, vol. 21, no. 1-3, pp. 1–6, 1998.

[5] V. Kvasnička, L. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Král', "Úvod do teórie neurónovỳch sietí," Iris, 1997.

[6] Haykin, Simon, "Self-organizing maps," 1999. `https://pdfs.semanticscholar.org/65db/2701ea14c4c127216a499d478015e1010591.pdf`.

[7] Kohonen, Teuvo, "Intro to SOM," 2005. `http://www.cis.hut.fi/projects/somtoolbox/theory/somalgorithm.shtml`.

[8] T. Kohonen, *Self-organization and associative memory*, vol. 8. Springer Science & Business Media, 2012.

[9] A. Baader and G. Hirzinger, "A self-organizing algorithm for multisensory surface reconstruction," in *Intelligent Robots and Systems' 94.'Advanced Robotic Systems and the Real World', IROS'94. Proceedings of the IEEE/RSJ/GI International Conference on*, vol. 1, pp. 81–88, IEEE, 1994.

[10] M. Strickert and B. Hammer, "Merge som for temporal data," *Neurocomputing*, vol. 64, pp. 39–71, 2005.

[11] B. Hammer, A. Micheli, A. Sperduti, and M. Strickert, "Recursive self-organizing network models," *Neural Networks*, vol. 17, no. 8, pp. 1061–1085, 2004.

[12] T. Voegtlin, "Recursive self-organizing maps," *Neural Networks*, vol. 15, no. 8, pp. 979–991, 2002.

[13] M. Johnsson, C. Balkenius, and G. Hesslow, "Associative self-organizing map.," in *IJCCI*, pp. 363–370, 2009.

[14] D. DeSieno, "Adding a conscience to competitive learning," in *IEEE international conference on neural networks*, vol. 1, pp. 117–124, IEEE Piscataway, NJ, 1988.

[15] D. E. Rumelhart, J. L. McClelland, P. R. Group, *et al.*, *Parallel distributed processing*, vol. 1. IEEE, 1988.

[16] P. A. Estévez and R. Hernández, "Gamma som for temporal sequence processing," in *International Workshop on Self-Organizing Maps*, pp. 63–71, Springer, 2009.

Appendix A

Attached CD contains source code for all experiments as well as all results.