ini uni | eth | zürich

Institute of Neuroinformatics

Realtime Topology Learning

Semester Thesis WS07/08

Student:
Matthias Schrag

Supervisors:
Dr. Kynan Eng
Dr. Tobi Delbruck

Abstract

Data gathered from a sensor network reflect its topology. It is therefore possible under certain circumstances to extract the topology later on from the data.

We developed an algorithm that calculates the adjacency of the network in realtime from timed events. It is based on a neural network with a spike-time dependent plasicity rule.

We implemented it a Java-based framework, so it can learn the topology from arbitrary connected devices like a neuromorphic vision chip. The learning process is monitored in realtime.

The algorithm is able to learn almost all adjacency relations of the chip. The quality of the result depends on the choice of timing parameters.

Contents

Chapter 1

Introduction

In an ad hoc sensor network the topology is not known from the beginning. But the knowledge of its topology is necessary for the interpretation of the data collected by it.

Martin Boerlin showed in [3] that it is possible to exploit dependencies in the data to guess the adjacency relations between the nodes of a network and therefore reconstruct its topology. He applied his algorithm to data from tactile floor [4] of the robot space 'Ada' [1] and also from a neuromorphic vision chip.

In this work Boerlin's solution is adapted for weak realtime. The adjacency is calculated incrementally, while the data is gathered from the network.

The sensor data is a sequence of events. Each event is given by a label, a timestamp and its data load. The label indicates the sender node, at which the event has been evoked, and the timestamp holds the time, when the event has been evoked. The data load in our case is just a type flag, that says whether the sensor nodes state switched to ON or to OFF.

Our algorithm extracts the topology information incrementally from the event data and guesses the adjacency relations of the sensor nodes.

A spiking neural network is used to perform this task. In this context our sensor data representation can be interpreted as address-event representation (AER): The idealized spikes in the neural network are represented by their address and their time. So the events are modelled as spikes in a neural network.

jAER [2] is a Java-based middleware for AER devices. jAER provides specific extensions of the events like the additonal type flag for ON and OFF events. Using this middleware we implemented our algorithm in Java and added a view to monitor the learning process in realtime.

We will first explain the problem and then we present our solution. We will give an overview on the implementation and its use. Then we will have a look at the results and end with a conclusion.

Chapter 2

Problem

Let us try to describe the problem formally: We have a set of events given by the sender node, a timestamp and the type of the event. The goal is, to learn the adjacency relations between the nodes.

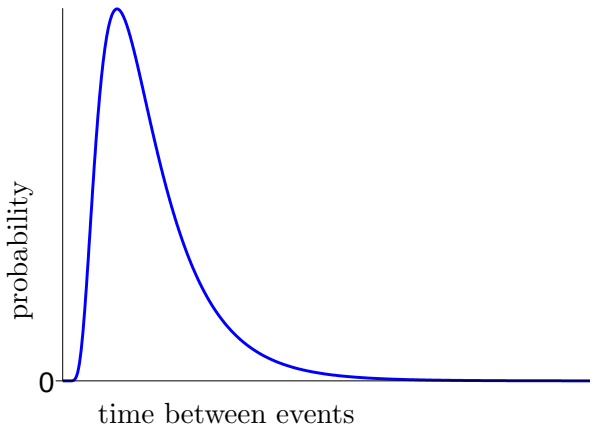
The events the nodes send are evoked by stimuli. When such a stimulus moves, it evokes events at the sensor nodes at different times. Since the stimulus does not make any jumps, there is a relationship between the times and the locations.

Let us first consider a single pair of events that are evoked by a stimulus moving from one node to the other. Since we do not know how it moves, we suppose that its motion is a Brownian motion. So the stimulus just moves randomly over the sensor network. Then the time the stimulus needs to evoke the two events is inversely Gaussian distributed [9] (see Figure 2.1).

The inverse Gaussian distribution is given by its probability density function

$$p(x;\lambda,\mu) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left(\frac{-\lambda(x-\mu)^2}{2\mu^2 x}\right) \tag{2.1}$$

μ is the distributions mean, λ is the so called shape parameter. The variance is calculated by μ^3/λ. You find a detailed explanation with examples in [8].



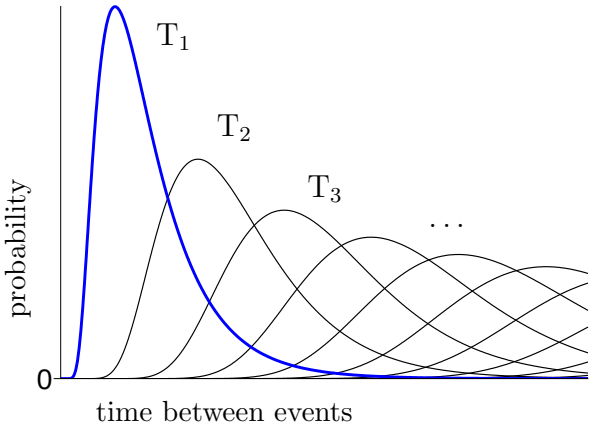
$\mathrm{T} \sim IG(\mu, \lambda)$

T : time between events
μ : mean
λ : shape parameter

Figure 2.1: Time distribution for a stimulus moving to the neighbor node.

While the stimulus moves further it reaches the next node - the neighbor's neighbor:

the so-called 2nd-order neighbor. The time it needs to reach this level is distributed similarly, as for the other higher order neighbors (see Figure 2.2).

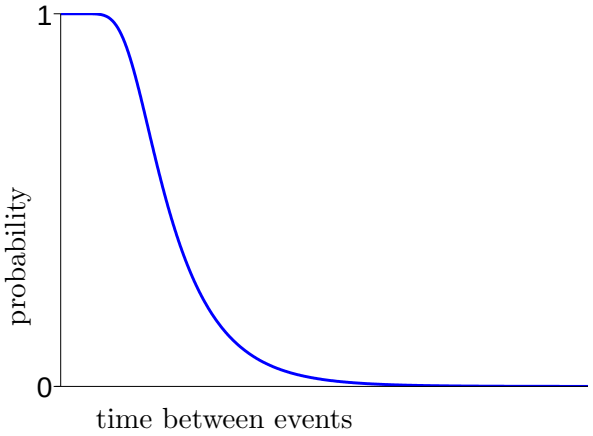


$$T_i \sim IG(\mu_i, \lambda_i)$$

T_i : time between events
$\mu_i = i\mu$: mean
$\lambda_i = i^2\lambda$: shape parameter

Figure 2.2: Time distribution to reach higher order neighbors.

To reach heigher order neighbors, the stimulus needs more time in average ($T_2, T_3, \ldots$). But there is an overlap in the distributions. It is not possible to distinguish a direct neighbor from a higher order neighbor by just looking at the time difference of a pair of events.
Possibly we would say that a node cannot be the first order neighbor if we just found the neighbor before with smaller time difference.

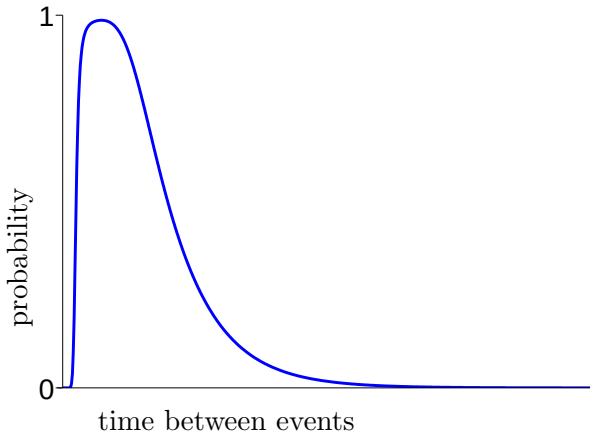


$$C_1 = \lim_{n\to\infty} \sum_{i=1}^{n} \frac{T_1}{T_i} \tag{2.2}$$

Figure 2.3: Probability of adjacency depending on time difference

So even if there was only a single stimulus in the system, it could not be decided from the time difference between two events, if the nodes are neighbors or higher order neighbors. Figure 2.3 shows the probability that the nodes the events are sent by are direct neighbors and not higher order neighbors. It assumes that the events are evoked by the same stimulus.
For very small time differences the adjacency is likely to be true.

If there is more than one stimulus in the system - what is mostly the case - there is also a certain probability that the events are not evoked by the same stimulus. This



$$C_\alpha = \lim_{n\to\infty} \sum_{i=1}^{n} \frac{\alpha T_1}{\alpha T_i + (1-\alpha)} \tag{2.3}$$

α : rate of events evoked by same stimulus

Figure 2.4: Adjusted probability of adjacency depending on time difference.

probability must be calculated from the number of stimuli. The resulting curve (Figure 2.4) is similar to the previous curve (Figure 2.3), but it tends to zero for small time differences, because its unlikely that a stimulus moves that fast: its much likelier that the two events are evoked by different stimuli.

Chapter 3

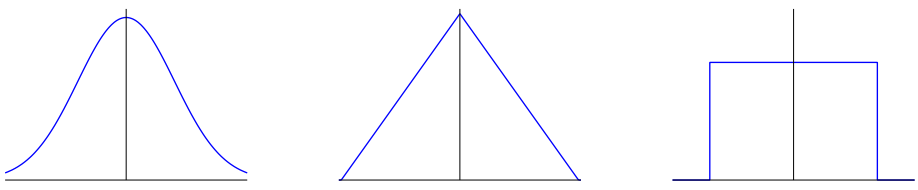
Algorithm

We exploit the non-unform distribution described above to learn the adjacency relationship in real-time as the events are coming in. The algorithm guesses for each pair of events the probability, that their nodes are adjacent. Instead of deciding, whether they actually are, it sums up all these probability values for each potential adjacency relation between two nodes. The idea is that real adjacency relations will get the biggest sums and the adjacency relations can be guessed. For a sufficiently large set of events, the guessed adjacency will converge to the real one.

A spiking neural network serves as model for the algorithm. The nodes of this network represent the sensor nodes, the edge weights between two nodes indicate, how likely the nodes are adjacent. The weights change according to a spike-time dependent plasticity rule (see Gerstner/Kistler, Spiking Neuron Models [5]).

Note that in our case the weight change does also depend on the type of the events. The adjacency relationship is symmetric, it would have been sufficient to have undirected edges. But for the purpose of extensibility and flexibility they are directed: the weight is changed only for a positive time difference; for negative ones it remains. The effect of this specification is that the edge weight changes only on a stimulus moving in the edges direction. Of course the symmetric case can be chosen expicitly - this is an option in our implementation.

The adjacency estimating function is therefore modeled as the learning window in our spiking neural network. The optimal learning window would look like in Figure 2.4. Since we do now know the parameters of this curve, and its calculation could be expensive, the following three shapes are used: a Gaussian, isosceles triangular function and a rectangular function (see Figure 3).



(a) Gaussian (b) Triangle (c) Rectangle

Figure 3.1: The three shape types for the learning window.

To make the shapes comparable, they are parametrized by the mean and standard

deviation. The standard deviation of the triangular function is $1/(2\sqrt{6})$ of the width of the learning window; for the rectangular it is $1/(2\sqrt{3})$.

For efficiency sake the Gaussian function is cut off outside $[-2.5\sigma, +2.5\sigma]$, so 99.7% of the shape lie inside this border. Further complexity reduction is achieved by caching the values of the Gaussian in a lookup table.

3.1 Steps of the algorithm

3.1.1 Initialization

At the beginning, all edge weights are initialized to 1 and the neighbors in the current guess (see 3.1.5) are chosen randomly.

3.1.2 Buffering

Since the learning window is of finite width[1], there is no need to store all the events coming in from the sensor network: a circular buffer of sufficient size is enough. The events are stored ordered by their timestamp. After an event has been processed, it will reach the end of the circular buffer and simply be overwritten by new events.

3.1.3 Pair selection

Before an event is stored in the buffer, it is compared with all the events whose timestamp lie within the the learning window of the event. Since the weight change is zero for negative values in the event window, those events are all contained in the buffer.

The nodes are of course uniquely labelled. So events coming from the same node are left out; only events from different nodes are selected.

3.1.4 Weight update

The weight change does not only depend on the difference between the timestamps of the events but also the event types: in our model there are two types of events indicating a direction – either to *on* or to *off*. So the learning window is more complicated than usual:

$$\Delta w_{ij} = \begin{cases} f(t_i - t_j) & d_i = d_j \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

i	postsynaptic node
j	presynaptic node
Δw_{ij}	weight change on edge from j to i
t_n	timestamp of event from node n
d_n	type of event from node n
$f : \mathbb{R}^+ \to \mathbb{R}^+$	learning window

The current weight change procedure has no negative feedback: the weights grow perpetually. In our tests this was not a problem. It has a positive impact on the algorithmic complexity, because the set of supposed neighbors of a node can be updated in constant time.

[1] For the gaussian shape, values outside 2.5σ are ignored.

Boerlin [3] in his algorithm normalized the weight vectors to unit size: this prevented the weights from perpetual growth. This step was left out because it would have required for each event pair to change the weights on the edges to all other nodes, not only between the two. Note that despite the symmetry of the adjacency relationship, the weight matrix which gives a guess for the adjacency is not symmetric any more as soon as the weights are normalized.
Since this is necessary also for the ability of adaptation, it will be an issue for later improvements of the algorithm. One could i.e. easily add a logarithmic fall back by storing to every edge weight the time of last access.

Boerlin's algorithm also reinforced the weights to the nodes in the current guess of neighbors of the node. This option can also be chosen in our implementation. But as there is currently no negative feedback, the algorithm gets stuck in a suboptimal solution.

$$\Delta w'_{ij} = \Delta w_{ij} + \begin{cases} \eta & j \in \hat{\mathbf{a}}_i \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

$\hat{\mathbf{a}}_i$ the nodes guessed to be adjacent to i (see below).

3.1.5 Adjacency guess

The edges with the biggest weights are supposed to lead to the neighbors of a node. So every node maintains a list with these supposed neighbors, and updates it each time an edge weight is updated.
In our implementation the correctness of this guess is monitored and displayed on an OpenGL device.

Chapter 4

Implementation

The topology learning algorithm is implemented in the jAER framework, a java frontend for camera devices using address-event representation.

Using jAER framework the topology learning is implemented as a subclass of `EventFilter2D` called `TopologyTracker`. The `TopologyTracker` gets the events in a method named `filterPacket` which is sent to all filters in the filter chain. Each event in the `EventPacket` is stored in a circular buffer. The buffer is implemented by a separate array for both fields of the event: the label which is built from the address, the timestamp and the type.

The current event is compared with the other events in the the buffer and the weight matrix is updated according to the learning window chosen. If the events are of the same type (OFF or ON) and do not have the same address, then the time difference is taken to update the weights. If the plastisity update function is symmetric (because the adjacency relationship is symmetric as well), then the function is mirrored.

It can be chosen between different shapes of this function: a rectangle, triangle and a gaussian function. They can be parametrized by the mean and standard deviation. For efficiency purpose the gaussian function is cut off at the border at 2.5 sigma, so 99.7% of the area is covered. One could think of cutting off more or less of the border depending on the actual realtime performance.

4.1 Restrictions of the Algorithm

The weighted edges connect each node with any other in the network. The size of the weight matrix is therefore quadratic to the number of nodes. And our implementation uses single precision floating point numbers as edge weights of the network. They occupy each 4 bytes in memory. So – n being the number of nodes – the memory used for the weight matrix is $4 \cdot n^2$, which is for the test data $4(2^{7 \cdot 2})^2$ Bytes = 1 GB.

So we downsized the input data by using only the inner 64x64 pixels. The matrix of now $4(2^{6 \cdot 2})^2$ Bytes = 64 MB easily fit into memory.

Other data structures do not need much memory. So the total memory needed is $\approx 4n^2$ Bytes $\in \mathcal{O}(n^2)$.

4.2 Objects

The two main classes are the `TopologyTracker` class and the `Monitor` class. The later is contained as an inner class.

4.2.1 `TopologyTracker`

The `TopologyTracker` implements the algorithm into the jAER framework. It subclasses the `EventFilter2D`. The core algorithm is implemented in the method `filterPacket` which overrides the the method inherited from `EventFilter2D`. This method handles the events, which are given in an `EventPacket` as method parameter. This core method uses two helper methods: `calculateWeightChange` and `adaptWeight`. The one calculates the weight change for an event pair according to the selected learning window. The other then adapts the weight matrix and the current adjacency guess.

4.2.2 `Monitor`

The `Monitor` supervises the topology learning process and displays it on the screen. It measures the performance and could also be extended to the algorithm parameters like the learning window to the current processing load to reach real-time performance.

In our case it knows the topology of the sensor in ahead - unlike the algorithm - and compares the current adjacency guess with the correct adjacencies, and it shows up the false adjacencies in the current guess and the ratio of correct ones.

Chapter 5

Running the Algorithm

Our algorithm, incarnated by the `TopologyTracker` is loaded automatically into the jAER Viewer's filter chain. Like any other filter we can activate or deactivate and reset it easily in the GUI. There are parameters to set (see Figure 5), that influence the learning process:

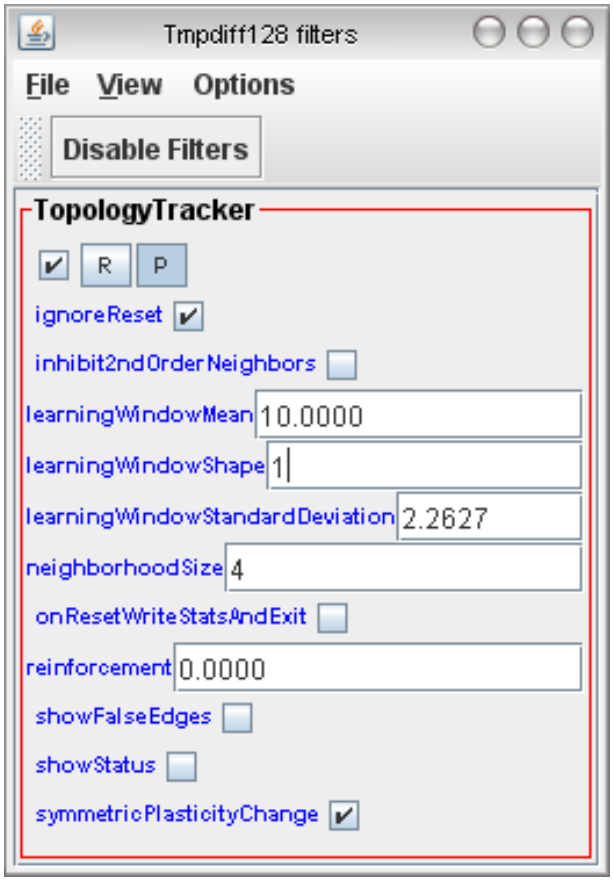


Figure 5.1: The parameters to set in jAER

- The learning window must be chosen depending on the input data
  - `learningWindowMean` - The mean time difference between two events in the learning window.

  - `learningWindowStandardDeviation` - The standard deviation of the time differences, defining the spread of the window
  - `learningWindowShape` - One of 3 shape types are provided: a Gaussian (0), a triangular function (1) and a rectangular function (2).

- `symmetricPlasticityChange` mirrors the learning window at $\Delta t = 0$, so the edge weights are changed symmetrically. This option is reasonnable because the adjacency relationship is symmetric as well - and it doubles the impact of any event.

- The neighborhood size - the number of adjacent nodes: e.g. 4 for a von Neuman neighborhood; 8 for a Moore neighborhood.

- There are two experimental extensions:

  - `inhibit2ndOrderNeighbors` let the algorithm ignore weight updates of edges between nodes which are second order neighbors according to the current adjacency guess.
  - `reinforcement`: A value > 0 reinforces the weights of the edges to the guessed neighbors of node by this amount, each time the node sends an event. This should speed up convergence, but does not yet work properly since the algorithm does not normalize the weights, so it gets stuck in a suboptimal solution.

- `showStatus` displays the algorithms status in a window: progress, utilization and false edges.

  - The false edges chart is only updated if `showFalseEdges` is on. One should switch it off at the beginning because it could otherwise overwhelm the graphics device.

The monitor panel looks as shown in Figures 5 and 5. It consists of the three subpanels *Progress*, *Utilization* and *Errors*.

Progress shows the current percentage of correctly guessed neighborship relations. The ordinate goes from 0 to 100%.

Utilization shows the current utilization, which is the processing time of an event packet relative to the time between the first and the last event in a packet. The values for graphs – *Progress* and *Utilization* – are calculated after each single packet and the curve of each graph is moved one step further.

Errors is a panel on which the false adjacency relations are displayed. Because of its vast need for processing time, updating can be inhibited by switching off `showFalseEdges`.

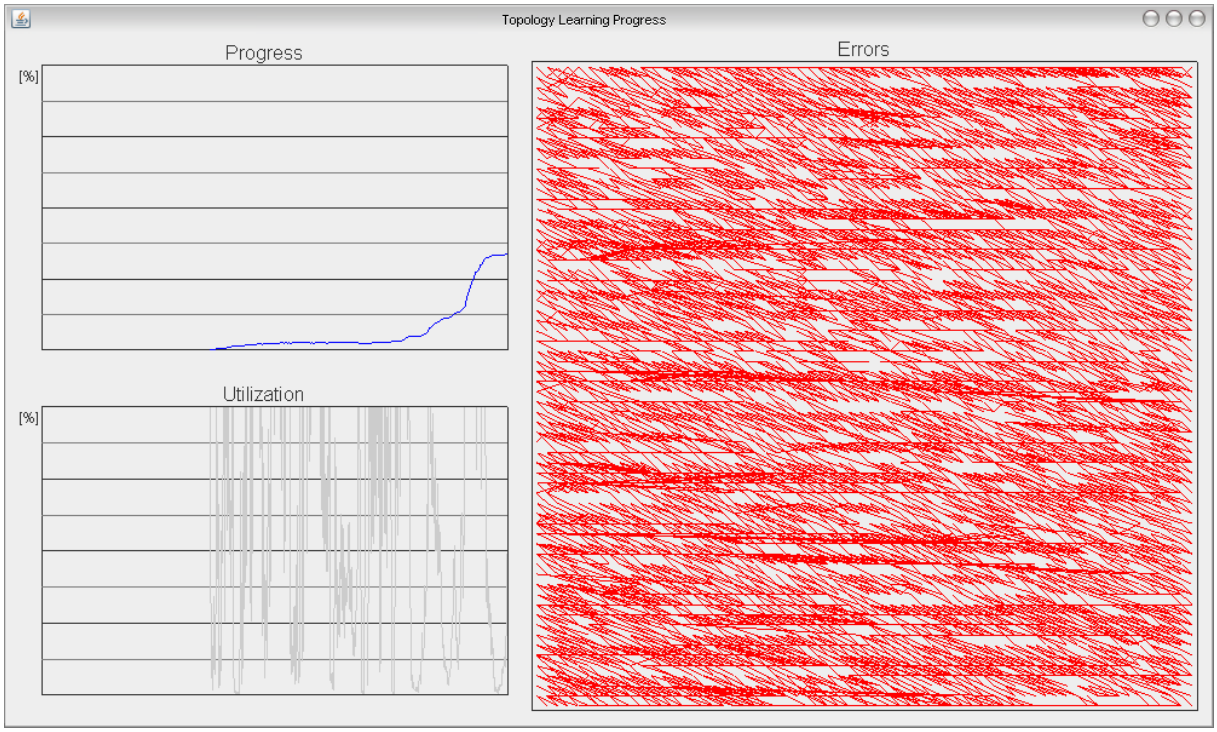


Figure 5.2: The monitor panel in an early state.

.

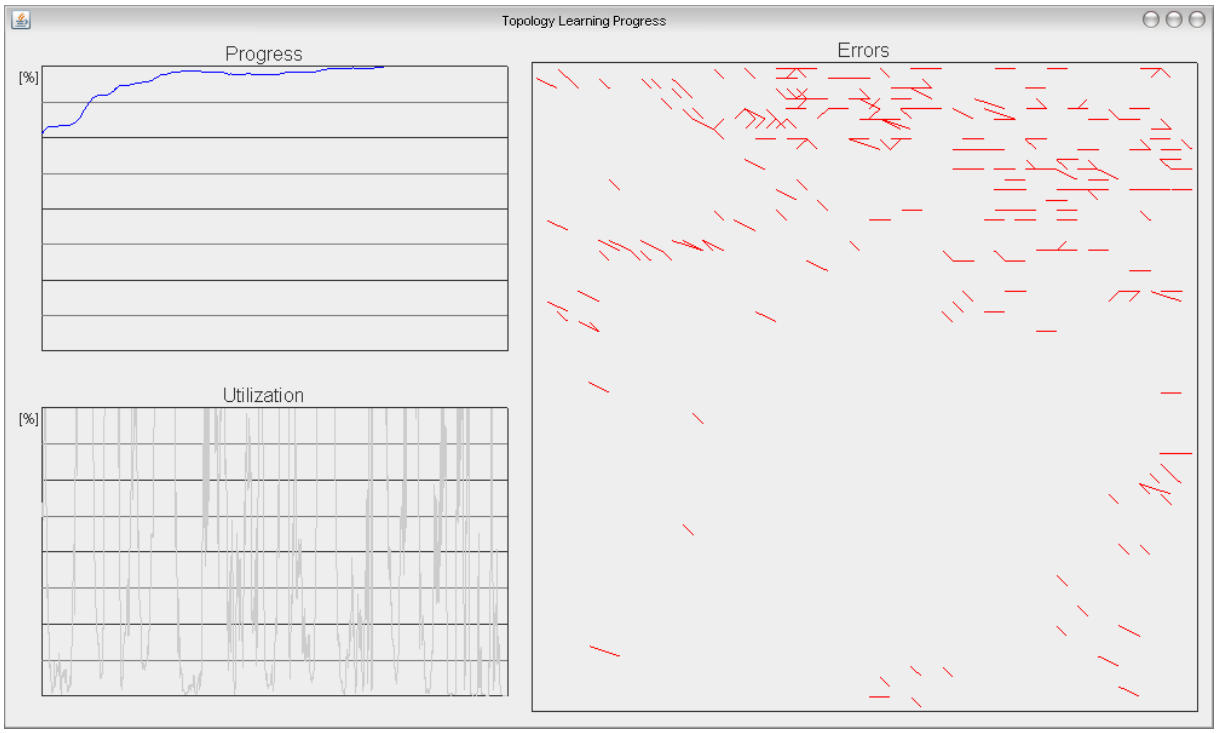


Figure 5.3: The monitor panel later on.

.

Chapter 6

Results

We tested our algorithm on a PC with Intel Core Duo T2300 Processor at 1.66 GHz and 1GB RAM. We used two different data sets of a resolution of 128x128 pixels.

6.1 Parameter fitting

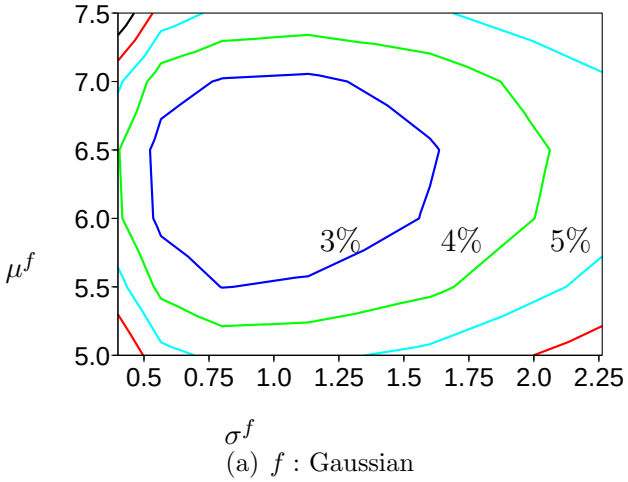
To get good results with the algorithm, the learning window must be chosen manually for each data. We tested three different shapes, which were parametrized by mean and standard deviation to make them comparable.

The three shapes differ in their comutational complexity: The rectangle needs no calculation after the selection of events, whereas the Gaussian is more costly to calculate. But since its function values can be cached in an array, there is no significant loss of speed.

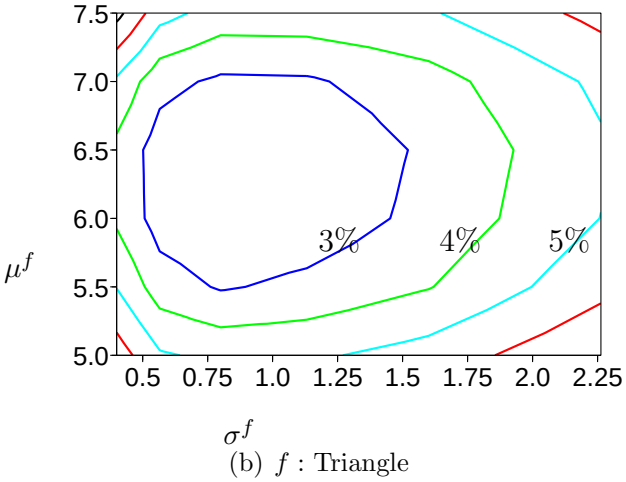
6.2 Remaining error

The following plots show the maximum ratio of correctly guessed adjacencies depending on the parameters of the learning window.

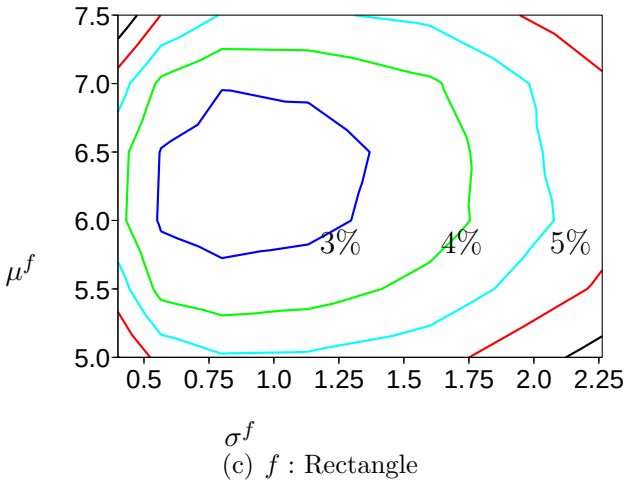
Both the Gaussian and the triangular function minimize the error to 2%. The rectangular reaches only 2.5% with the given data. This is very similar. But the Gaussian is slightly better than the others.



(a) f : Gaussian



(b) f : Triangle

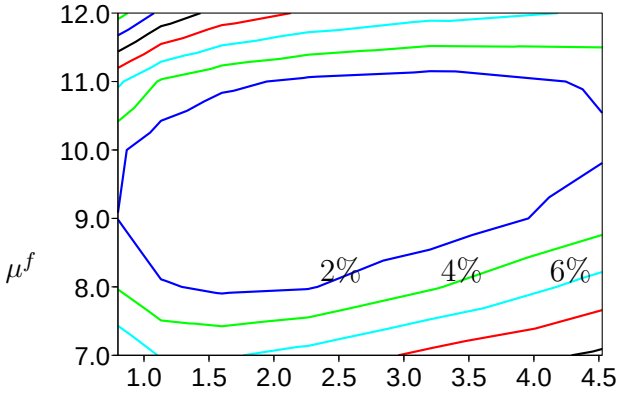


(c) f : Rectangle

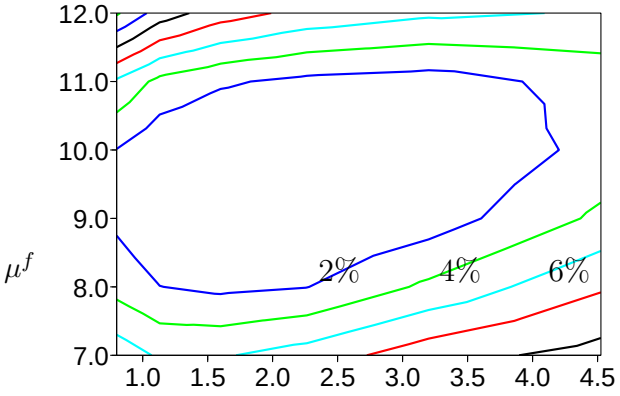
Figure 6.1: Data set 1: Remaining error for different shapes parametrized by μ and σ. The curves are at 3%, 4%, 5%... errors.

All the three shapes here can eliminate almost all of the errors. But more obvious than in the plots before, the rectangular shape is less stable than the other two. Best is always the Gaussian.

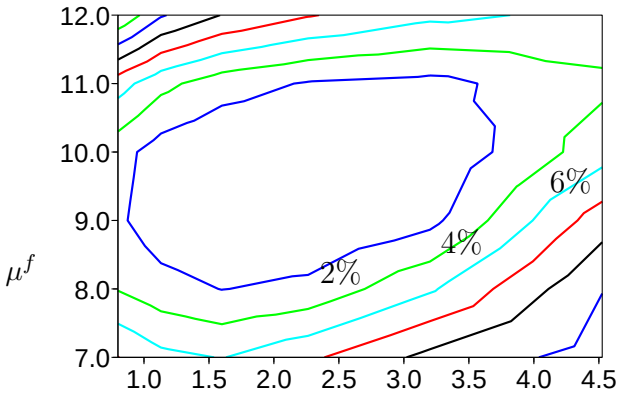
The Gaussian algorithm can estimate almost all the adjacency relations correctly. The problem is, that this depends on the previous parameter setting. When the data changes its behaviour over time, this good result is lost. For the data set 2 paramters μ and σ can be set in a way, that the algorithm finds almost all the adjacency relations after half the time. But since the data changes, those parameters suddenly are not appropriate any more and the primary correct guess gets worse.



(a) f : Gaussian



(b) f : Triangle

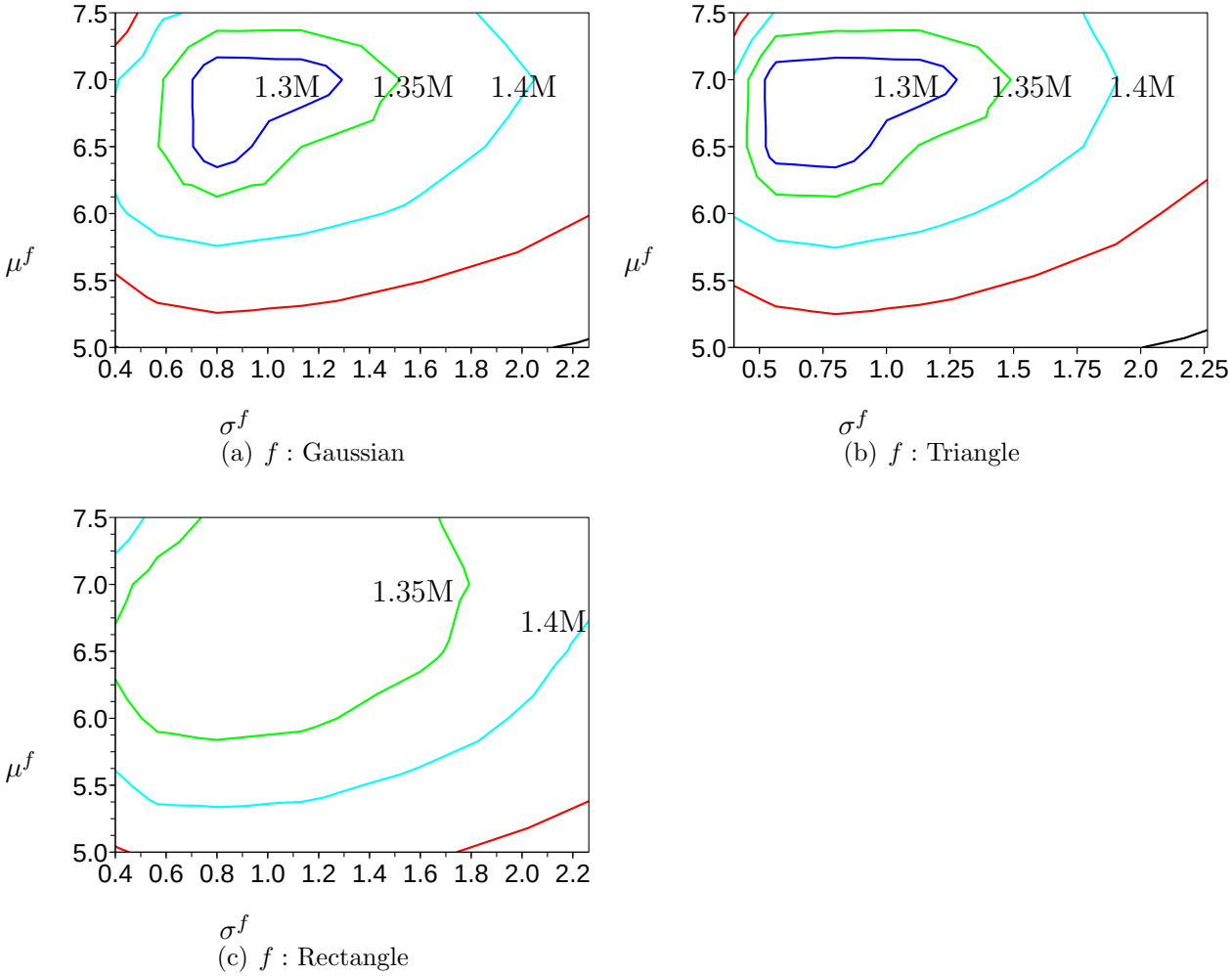


(c) f : Rectangle

Figure 6.2: Data set 2: Remaining error for different shapes parametrized by μ and σ. The curves are at 2%, 4%, 6%... errors.

6.3 Number of events

Also the number of events needed to reach a certain level of correctness differs. We set the level at 7/8 of the correctly guessed adjacency relations.



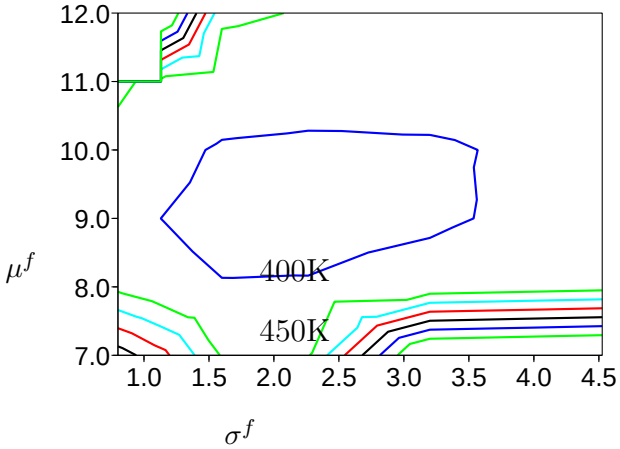
(a) f : Gaussian

(b) f : Triangle

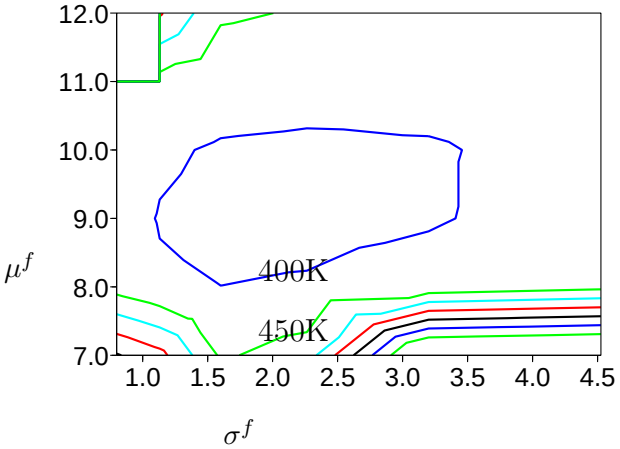
(c) f : Rectangle

Figure 6.3: Data set 1: Number of events needed to guess 7/8 of the adjacencies correctly. The curves are at 1.3M, 1.35M, 1.4M... events.

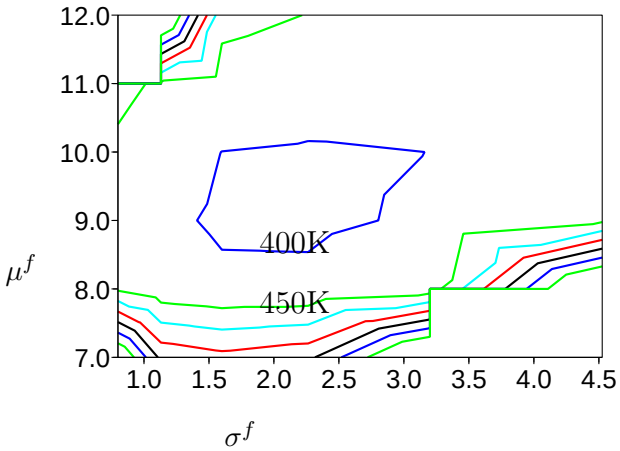
This plots show clearlier, that the rectangular shape is less performant than the others: The Gaussian and triangular function need only 1.3 million events whereas the rectangular needs 1.4 million.



(a) f : Gaussian



(b) f : Triangle



(c) f : Rectangle

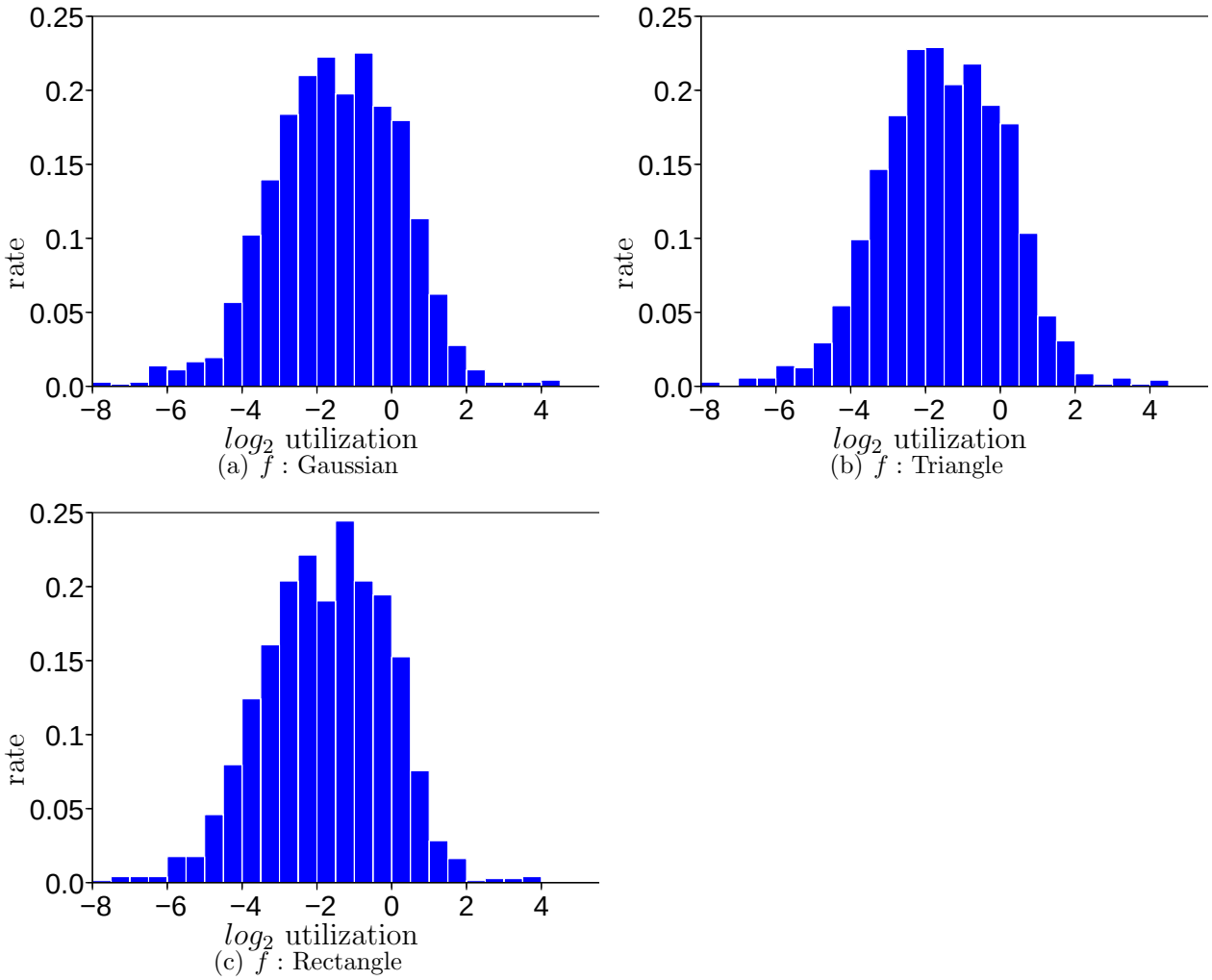
Figure 6.4: Data set 2: Number of events needed to guess 7/8 of the adjacencies correctly. The curves are at 400K, 450K, 500K events.

In the second data set the rectangle needs as best result only 1% more events than the other shapes, but the other shapes are more stable. As you can see, the rectangle with mean at 7 ms and the huge variance does not even let the algorithm reach the level of 7/8 correct adjacency relations: this edge is cut off from the graph. The Gaussian and triangular shape on the other hand can cope even with this extremal values.

6.4 Time

The processing time is linear to the learning window size. The smaller the chosen learning window is, the faster the data is processed. As the dependency is only linear, there is no need to keep the learning window small.

The following figure shows a histogram plot of the utilization for the adjacency estimation based on the data set 1 with a Gaussian with $\mu = 6.5$ms and $\sigma = 0.8$ms. The utilization is calculated by division of the time needed to process an event packet by the timespan between the timestamp of the processed events in the packet.



(a) f : Gaussian

(b) f : Triangle

(c) f : Rectangle

Figure 6.5: Utilization of the computer

Figure 6.4 show histogram plots of the logarithm of the utilization values (logarithm to the base of 2). The data set 1 with $\mu = 6$ms and $\sigma = 0.8$ms has been taken. Since the Gaussian curve is cached in a lookup table, the three histograms do not differ much.
But we can see that the utilization sometimes is bigger than 1: its logarithm is more than 0. This exceeding part is 14% for the rectangular shape, 19% for the triangle and 20% for the Gaussian. It higly depends on the chosen parameters.

To improve the realtime performance of the algorithm, one could change it to ignore any events as the utilization tends towards 1. We already tried out a simple version of

such an extension. But it seems, that a utilization prediction is needed to overcome this issue.

Chapter 7

Conclusion

As we have seen, the algorithm presented gives us the ability to learn the adjacency relations of the nodes in a sensor network, while the data is gathered. And with a Gaussian or triangular function as shape of the learning window, the error rate of the estimation is quite small even if their parameters are not set precisely. The rectangular shape does not bring equally good results.
Our implementation in Java monitors the learning process in realtime. So there is a tool, which lets us study the learning performance for different parameters and which can also easily be extended.

There are several issues left, that are not yet addressed in the current algorithm. The learning window, so the weight updating rule, must be parametrized in ahead: The algorithm does not adapt to the data. It would be great to have such an adaptive algorithm. This is probably possible by checking the input data against the model guessed from the previous data.
The algorithm does in general not use the model to improve the estimation process – apart from this experimental reinforcement. And it is also only a very simple model. One could imagine to use the weight matrix to really construct the topology. I presume that such an abitration of the neighbors, which depends on guess of the overall topology, would lead to much better results.
There is also the issue of perpetual growing weights left. A logarithmic fall back would be nice in terms of the algorithmic complexity, but would possibly lead to poor results. Maybe mit would be feasible to implement a rule that normalizes the weights implicitly, like Oja's rule [7] (for rate based learning) or the Riccati rule for spiking neurons [6].
The algorithm does not adapt itself to the realtimeness: it should be easy to change the domain of the learning window function upon realtime performance.

This work once more was instructive to me: it gave me a chance to get a glimpse of what other engineers do and how I can contribute. A also learnt a lot on neural network models and realtime visualization in OpenGL. I would like to thank Kynan Eng and Paul Rogister for helping me and guiding me through this work.

Bibliography

[1] Ada – the intelligent space. See URL http://ada.ini.ethz.ch.

[2] jaer open source project. See URL http://jaer.wiki.sourceforge.net.

[3] Martin Boerlin. Getting to know your neighbors: reconstructing topology from real-world input. 2006.

[4] Tobi Delbrück et al. A tactile luminous floor for an interactive autonomous space. *Robotics and Autonomous Systems*, 55(6):433–443, 2007.

[5] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[6] Philipp D. Häfliger. *A Spike Based Learning Rule and its Implementation in Analog Hardware*. PhD thesis, Swiss Federal Institute of Technology, Zürich, 2000.

[7] Erkki Oja. A Simplified Neuron Model as a Principal Component Analyzer. *Journal of Mathematical Biology*, 15:276–273, 1982.

[8] V. Seshadri. *The inverse Gaussian distribution: Statistical theory and applications*. Lecture Notes in Statistics. Springer, 1998.

[9] The Free Encyclopedia Wikipedia. Inverse Gaussian distribution, 2008.

Appendix A

The Code

```
/*
 * TopologyTracker.java
 *
 * contains the classes TopologyTracker and TopologyTracker.Monitor.
 * A TopologyTracker makes a guess of the adjacency matrix of the pixels
 * by tracking the events coming from the pixels.
 * The monitor is used to display the algorithms progress and current state.
 *
 * Semester project Matthias Schrag, HS07
 */

package ch.unizh.ini.caviar.eventprocessing.tracking;

import ch.unizh.ini.caviar.chart.Axis;
import ch.unizh.ini.caviar.chart.Category;
import ch.unizh.ini.caviar.chart.VectorFieldChart;
import ch.unizh.ini.caviar.chart.VectorSeries;
import ch.unizh.ini.caviar.chart.XYChart;
import ch.unizh.ini.caviar.chip.AEChip;
import ch.unizh.ini.caviar.event.BasicEvent;
import ch.unizh.ini.caviar.event.EventPacket;
import ch.unizh.ini.caviar.event.TypedEvent;
import ch.unizh.ini.caviar.eventprocessing.EventFilter2D;
import ch.unizh.ini.caviar.chart.Series;
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Observable;
import java.util.Observer;
import java.util.Random;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**
 *
 * @author Matthias
 */
public class TopologyTracker extends EventFilter2D implements Observer {

    protected static final int DEFAULT_NEIGHBORHOOD_SIZE = 4;
    protected static final int DEFAULT_MAX_SQUARED_NEIGHBORHOOD_DISTANCE = 1;
    protected static final float DEFAULT_LEARNING_WINDOW_CENTER = 5.0f; // 5 ms per default
    protected static final float DEFAULT_LEARNING_WINDOW_STANDARD_DEVIATION = 1.0f; // 1 ms standard deviation

    protected static final int MAX_SIZE_X = 64;
    protected static final int MAX_SIZE_Y = 64;
    protected static final int BUFFER_SIZE = 10000; // the number of events in event window
    protected static final int NO_LABEL = -1;
    protected static final int NO_TIMESTAMP = Integer.MIN_VALUE;
    protected static final byte NO_TYPE = -1;
    protected static final float INITIAL_WEIGHT = 1.0f;
    protected static final float BELL_CURVE_SIZE = 2.5f;
    protected static final int BELL_CURVE_RESOLUTION = 1000;
    protected static final float[] BELL_CURVE = new float[(int) (BELL_CURVE_SIZE * BELL_CURVE_RESOLUTION)];   // the Gaussian in range [ 0

    /**
     * Event source data structures.
     */
    protected AEChip chip;
    protected int sizeX, sizeY; // needed for labelling
    protected int minX, minY;   // needed to filter out subrange
    protected int maxX, maxY;   // needed to filter out subrange

    /**
     * The event window with fixed events count <code>windowSize</code>.
```

A The Code

```
 * An event consists of its source and timestamp, both stored in simultaneous arrays.
 * <code>eventsIndex</code> denotes the index of the current event.
 */
protected int[] eventsSource;    // the labels of the nodes in event window
protected int[] eventsTimestamp;    // the timestamp of the events in event window
protected byte[] eventsType;    // the type of the event: ON|OFF
protected int eventWindowBegin = 1;
protected int eventWindowEnd = BUFFER_SIZE;
protected int eventIndex;   // the index of the inserted event
protected long current; // the number of events passed

/**
 * Algorithm parameters.
 */
protected int learningWindowShape;
protected float learningWindowMean;
protected float learningWindowStandardDeviation;
protected int neighborhoodSize;
protected boolean inhibit2ndOrderNeighbors;
protected boolean symmetricPlasticityChange;
protected float reinforcement;
private boolean ignoreReset;

/**
 * Monitor parameters.
 */
protected int maxSquaredNeighborDistance;
protected boolean showStatus;
protected boolean showFalseEdges;
protected boolean onResetWriteStatsAndExit;

/**
 * Algorithm data structures.
 */
protected float[][] weights;  // the adjacency guess
protected int[][] neighbors;  // the current neighbors guess
protected float learningWindowBegin;
protected float learningWindowEnd;

/**
 * The monitor and stat data.
 */
protected Monitor monitor;
protected Vector<String> params;
protected Vector<String> stat;
protected ArrayList<Float> utilizationStat;
protected long startTime;
protected JFrame window;

/**
 * Create a new TopologyTracker
 */
public TopologyTracker(AEChip chip) {
    super(chip);
    this.chip = chip;
    initFilter();

    /* read alogorithm parameters */
    neighborhoodSize = getPrefs().getInt("TopologyTracker.neighborhoodSize", DEFAULT_NEIGHBORHOOD_SIZE);
    setPropertyTooltip("neighborhoodSize", "Number of neighbors a single pixel can have.");
    maxSquaredNeighborDistance = getPrefs().getInt("TopologyTracker.maxSquaredNeighborDistance", DEFAULT_MAX_SQUARED_NEIGHBORHOOD_DIST
    setPropertyTooltip("maxSquaredNeighborDistance", "[monitor] Maximum squared distance between two neighbors. If neighborhoodSize is

    learningWindowShape = getPrefs().getInt("TopologyTracker.learningWindowShape", 2);
    setPropertyTooltip("learningWindowShape", "Shape of the learning window [0=Gaussian 1=Triangle 2=Rectangle].");
    learningWindowMean = getPrefs().getFloat("TopologyTracker.learningWindowMean", DEFAULT_LEARNING_WINDOW_CENTER);
    setPropertyTooltip("learningWindowMean", "Center of the learning window [ms].");
    learningWindowStandardDeviation = getPrefs().getFloat("TopologyTracker.learningWindowStandardDeviation", DEFAULT_LEARNING_WINDOW_S
    setPropertyTooltip("learningWindowStandardDeviation", "Width of the learning window [ms].");
    updateLearningWindow();

    symmetricPlasticityChange = getPrefs().getBoolean("TopologyTracker.symmetricPlasticityChange", true);
    setPropertyTooltip("symmetricPlasticityChange", "Update also symmetric edge in adjacency matrix.");
    inhibit2ndOrderNeighbors = getPrefs().getBoolean("TopologyTracker.inhibit2ndOrderNeighbors", false);
    setPropertyTooltip("inhibit2ndOrderNeighbors", "Check before weight update if potential neighbor is guessed to be a 2nd-order neig
    reinforcement = getPrefs().getFloat("TopologyTracker.reinforcement", 0.0f);
    setPropertyTooltip("reinforcement", "Reinforce edge weights of neighbors in current guess.");
    ignoreReset = getPrefs().getBoolean("TopologyTracker.ignoreReset", true);
    setPropertyTooltip("ignoreReset", "Do not reset the filter upon reset events (but eventially write stats).");

    showStatus = getPrefs().getBoolean("TopologyTracker.showStatus", true);
    setPropertyTooltip("showStatus", "[monitor] Show the algorithms status in a frame.");
    showFalseEdges = false; // start without display of false edges
    setPropertyTooltip("showFalseEdges", "[monitor] Show the false edges in a diagram.");
    onResetWriteStatsAndExit = getPrefs().getBoolean("TopologyTracker.onResetWriteStatsAndExit", false);
    setPropertyTooltip("onResetWriteStatsAndExit", "[stat] Log stat data to file.");

    chip.addObserver(this);
    monitor = new Monitor();

    window = new JFrame("Topology Learning Progress");
    //window.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    window.setPreferredSize(new java.awt.Dimension(1000, 600));
    window.setLocation(200, 100);   // TODO
    window.setLayout(new BorderLayout());
    JPanel panel = new JPanel();
```

```
    panel.setLayout(new GridLayout(2, 1));
    panel.add(monitor.progressChart);
    panel.add(monitor.utilizationChart);
    window.getContentPane().add(BorderLayout.CENTER, panel);
    window.getContentPane().add(BorderLayout.EAST, monitor.vectorChart);
    window.pack();
    window.setVisible(showStatus);
}

/**
 * @see EventFilter.initFilter()
 * this should allocate and initialize memory:
 * it may be called when the chip e.g. size parameters are changed after creation of the filter
 */
public void initFilter() {
    /* set up constants */
    double x = 0.0f;
    for (int i = 0; i < BELL_CURVE.length; i++) {
        BELL_CURVE[i] = (float) (Math.exp(-x*x / 2) / Math.sqrt(2 * Math.PI));
        x += 1.0f / BELL_CURVE_RESOLUTION;
    }

    /* set range */
    if ((sizeX = chip.getSizeX()) > MAX_SIZE_X) {
        sizeX = MAX_SIZE_X; // limit sizeX
        minX = sizeX /2;
    } else {
        minX = 0;
    }
    maxX = minX + sizeX;
    if ((sizeY = chip.getSizeY()) > MAX_SIZE_Y) {
        sizeY = MAX_SIZE_Y; // limit sizeY
        minY = sizeY /2;
    } else {
        minX = 0;
    }
    maxY = minY + sizeY;
    if (sizeX * sizeY == 0) {
        System.err.println("Chip dimensions not found: cannot init TopologyTracker!");
        return;  // ERROR!
    }
    /* create model */
    int n = sizeX*sizeY;
    weights = new float[n][n];
    neighbors = new int[n][neighborhoodSize+1];    // last element is dummy
    // create event arrays
    eventsSource = new int[BUFFER_SIZE];
    eventsTimestamp = new int[BUFFER_SIZE];
    eventsType = new byte[BUFFER_SIZE];

    if (ignoreReset) reset();   // ignore reset events, so do it now...
}

/**
 * @see EventFilter.resetFilter()
 * should reset the filter to initial state
 */
public void resetFilter() {
    /* write stats */
    if (onResetWriteStatsAndExit) monitor.writeStat();

    if (ignoreReset) return;
    // body
    reset();
}

/**
 * Reset the filter.
 */
public void reset() {
    Arrays.fill(eventsSource, NO_LABEL);
    Arrays.fill(eventsTimestamp, NO_TIMESTAMP);
    Arrays.fill(eventsType, NO_TYPE);
    eventIndex = 0;
    Random random = new Random();
    int n = sizeX*sizeY;
    for (int i = 0; i < weights.length; i++) {
        /* init weight */
        Arrays.fill(weights[i], INITIAL_WEIGHT);
        /* init neighbors randomly */
        for (int rank = 0; rank < neighborhoodSize; rank++) {
            int j;
            int r;
            do {
                j = random.nextInt(n-1); if (j >= i) j++;   // random neighbor (without node itself)
                for (r = 0; r < rank && neighbors[i][r] != j; r++);    // check for collision
            } while (r < rank); // repeat while collision
            neighbors[i][rank] = j;
        }
    }
    monitor.init();
}

/**
 * @see EventFilter.getFilterState()
 * should return the filter state in some useful form
```

```
     * @deprecated - no one uses this
     */
    public Object getFilterState() {
        return null;
    }

    /**
     * Calculates the learning window begin and end.
     */
    protected void updateLearningWindow() {
        float halfWidth = learningWindowStandardDeviation;
        switch (learningWindowShape) {
            case 0: halfWidth = learningWindowStandardDeviation * BELL_CURVE_SIZE; break; // Gaussian (for 99.7%)
            case 1: halfWidth = learningWindowStandardDeviation * (float) Math.sqrt(6); break; // Triangle
            case 2: halfWidth = learningWindowStandardDeviation * (float) Math.sqrt(3); break; // Rectangle
            default:
        }
        learningWindowBegin = learningWindowMean - halfWidth;
        learningWindowEnd = learningWindowMean + halfWidth;
    }

    /**
     * Core method:
     * add new events to queue, learn topology
     */
    public EventPacket<?> filterPacket(EventPacket<?> in) {
        if (!filterEnabled) return in;
        if (enclosedFilter != null) in = enclosedFilter.filterPacket(in);
        assert in != null;

        monitor.enter(in.getSize(), in.getLastTimestamp() - in.getFirstTimestamp());    // begin monitored sequence

        for (BasicEvent event : in) {
            if (event.x < minX || event.x >= maxX || event.y < minY || event.y >= maxY) continue;   // treat only events inside borders
//            long start = System.nanoTime();

            /* create event (timestamp, label, type) */
            int t = event.timestamp;
            int i = (event.x - minX) * sizeY + (event.y - minY);
            byte type = (event instanceof TypedEvent) ? ((TypedEvent) event).type : NO_TYPE;
            current++;

            /* learn topology */
            int endIndex = (eventIndex + eventWindowEnd) % BUFFER_SIZE;
            // iterate over previous events...
            for (int index = eventWindowBegin; index < eventWindowEnd; index++) {
                int current = (eventIndex + index) % BUFFER_SIZE;
                if (eventsType[current] != type) continue;   // weight change only if events have same type
                // set aliases
                int timestamp = eventsTimestamp[current];
                float dt = (t - timestamp) / 1000.0f;  // convert from microseconds to milliseconds
                if (dt < learningWindowBegin) continue;
                if (dt >= learningWindowEnd) break;
                int j = eventsSource[current];
                if (i == j) continue;

                /* adapt weight */
                float weightChange = calculateWeightChange(weights[i][j], dt);
                // check if higher order neighbor
                if (inhibit2ndOrderNeighbors) {
                    for (int rank = 0; rank < neighborhoodSize; rank++) {
                        int direct = neighbors[i][rank];
                        for (int r = 0; rank < neighborhoodSize; rank++) {
                            if (neighbors[direct][r] == j) weightChange = 0.0f;
                        }
                    }
                }
                adaptWeight(i, j, weightChange);
                if (symmetricPlasticityChange) adaptWeight(j, i, weightChange);
                // reinforce winners
                if (reinforcement > 0) {
                    for (int rank = 0; rank < neighborhoodSize; rank++) {
                        int neighbor = neighbors[i][rank];
                        weights[i][neighbor] += reinforcement;  // adapt weight; does not disturb ranking of neighbors
                    }
                }
            } // improvement: adapt eventWindowSize to real time performance
            eventIndex = (eventIndex - 1 + BUFFER_SIZE) % BUFFER_SIZE;  // eventIndex-- (BUFFER_SIZE)

            /* replace oldest event by current event */
            eventsTimestamp[eventIndex] = t;
            eventsSource[eventIndex] = i;
            eventsType[eventIndex] = type;
        }

        monitor.exit();
        return in;
    }

    public int getNeighborhoodSize() {
        return neighborhoodSize;
    }

    public void setNeighborhoodSize(int value) {
        getPrefs().putInt("TopologyTracker.neighborhoodSize", value);
        support.firePropertyChange("neighborhoodSize", neighborhoodSize, value);
```

```
    neighborhoodSize = value;
}

public int getLearningWindowShape() {
    return learningWindowShape;
}

public void setLearningWindowShape(int value) {
    getPrefs().putInt("TopologyTracker.learningWindowShape", value);
    support.firePropertyChange("learningWindowShape", learningWindowShape, value);
    learningWindowShape = value;
    updateLearningWindow();
}

public float getLearningWindowMean() {
    return learningWindowMean;
}

public void setLearningWindowMean(float value) {
    getPrefs().putFloat("TopologyTracker.learningWindowMean", value);
    support.firePropertyChange("learningWindowMean", learningWindowMean, value);
    learningWindowMean = value;
    updateLearningWindow();
}

public float getLearningWindowStandardDeviation() {
    return learningWindowStandardDeviation;
}

public void setLearningWindowStandardDeviation(float value) {
    getPrefs().putFloat("TopologyTracker.learningWindowStandardDeviation", value);
    support.firePropertyChange("learningWindowStandardDeviation", learningWindowStandardDeviation, value);
    learningWindowStandardDeviation = value;
    updateLearningWindow();
}

public boolean getInhibit2ndOrderNeighbors() {
    return inhibit2ndOrderNeighbors;
}

public void setInhibit2ndOrderNeighbors(boolean value) {
    getPrefs().putBoolean("TopologyTracker.inhibit2ndOrderNeighbors", value);
    support.firePropertyChange("inhibit2ndOrderNeighbors", inhibit2ndOrderNeighbors, value);
    inhibit2ndOrderNeighbors = value;
}

public boolean getSymmetricPlasticityChange() {
    return symmetricPlasticityChange;
}

public void setSymmetricPlasticityChange(boolean value) {
    getPrefs().putBoolean("TopologyTracker.symmetricPlasticityChange", value);
    support.firePropertyChange("symmetricPlasticityChange", symmetricPlasticityChange, value);
    symmetricPlasticityChange = value;
}

public float getReinforcement() {
    return reinforcement;
}

public void setIgnoreReset(boolean value) {
    getPrefs().putBoolean("TopologyTracker.ignoreReset", value);
    support.firePropertyChange("ignoreReset", ignoreReset, value);
    ignoreReset = value;
}

public boolean getIgnoreReset() {
    return ignoreReset;
}

public void setReinforcement(float value) {
    getPrefs().putFloat("TopologyTracker.reinforcement", value);
    support.firePropertyChange("reinforcement", reinforcement, value);
    reinforcement = value;
}

public boolean getShowStatus() {
    return showStatus;
}

public void setShowStatus(boolean value) {
    getPrefs().putBoolean("TopologyTracker.showStatus", value);
    support.firePropertyChange("showStatus", showStatus, value);
    showStatus = value;
    window.setVisible(showStatus);
}

public boolean getShowFalseEdges() {
    return showFalseEdges;
}

public void setShowFalseEdges(boolean value) {
    getPrefs().putBoolean("TopologyTracker.showFalseEdges", value);
    support.firePropertyChange("showFalseEdges", showFalseEdges, value);
    showFalseEdges = value;
    if (showFalseEdges) {
```

```
        for (int i = 0; i < neighbors.length; i++) {
            monitor.neighborhoodChanged(i, neighbors[i]);
        }
    }
}

public boolean getOnResetWriteStatsAndExit() {
    return onResetWriteStatsAndExit;
}

public void setOnResetWriteStatsAndExit(boolean value) {
    getPrefs().putBoolean("TopologyTracker.onResetWriteStatsAndExit", value);
    support.firePropertyChange("onResetWriteStatsAndExit", onResetWriteStatsAndExit, value);
    onResetWriteStatsAndExit = value;
}

public void update(Observable o, Object arg) {
    initFilter();
}

/**
 * Change the weight(i,j) by a certain non-negative <code>amount</code>.
 *
 * Guess for neighbors (neighbors with highest weights) is recalculated.
 */
private void adaptWeight(int i, int j, float amount) {
    /* precondition */
    assert 0 <= i && i < weights.length;    // i valid node label
    assert 0 <= j && j < weights[i].length; // j valid node label
    assert amount >= 0;                     // amount non-negative

    /* raise weight by amount */
    weights[i][j] += amount;

    /* reestablish ranking among guessed neighbors of i */
    int[] neighbor = neighbors[i];
    float[] weight = weights[i];
    int rank;
    int higher, lower;
    neighbor[neighborhoodSize] = j;  // dummy at end
    for (rank = 0; neighbor[rank] != j; rank++);   // find node j in neighbors
    // re-sort neighbors in ranking
    while (rank > 0 && weight[ (lower = neighbor[rank-1]) ] < weight[ (higher = neighbor[rank]) ]) {
        neighbor[rank-1] = higher; neighbor[rank] = lower;    // swap rank of competing neighbors
        rank--;
    }   // post: forall j: weight[neighbor[j-1]] >= weight[neighbor[j]]
    if (rank < neighborhoodSize) { // if neighbor is new...
        monitor.neighborChanged(i, j, neighbor, rank);  // ...monitor correctness of new guess
        if (showStatus && showFalseEdges) monitor.neighborhoodChanged(i, neighbor);
    }
}

/**
 * Calculate a weight change depending on timestamp differences
 */
private float calculateWeightChange(float weight, float dt) {
    if (dt < 0) return 0.0f;
    float halfWidth = learningWindowMean - learningWindowBegin;
    switch (learningWindowShape) {
        case 0: // Gaussian
            float x = Math.abs((dt - learningWindowMean) / learningWindowStandardDeviation);
            int i = (int) (x * BELL_CURVE_RESOLUTION);
            if (i < BELL_CURVE.length) return BELL_CURVE[i] / learningWindowStandardDeviation;
            else return 0.0f;   // round to zero for dt - mu >= BELL_CURVE_SIZE*sigma (==2.5*sigma)
        case 1: // Triangle
            if (dt < learningWindowMean) return (dt - learningWindowBegin) / halfWidth / halfWidth;
            else return (learningWindowEnd - dt) / halfWidth / halfWidth;
        case 2: // Rectangle
            return 0.5f / halfWidth;
        default:
            return 0.0f;
    }
}

/**
 * The Monitor class.
 *
 * A monitor knows the structure of the pixels in ahead and can therefore display the progress.
 *
 * Remark: The outer class (TopologyTracker) does not access any non-public fields/methods.
 */
public class Monitor {

    public final int DISPLAY_FREQUENCY = 30;

    private int inputDuration;  // current time span of input timestamps
    private int referenceTime;  // current begin time in microseconds
    private int displayTime = Integer.MIN_VALUE;

    private float correctNeighbors;
    private float totalNeighbors;
    private float currentBest;
    private float outstanding;

    /** The monitor's view */
    public float time = 0;
```

```
public Series progress;
public Series utilization;
public VectorSeries[] rankedErrors;

private Category progressCurve;
private Category utilizationCurve;
private Category windowSizeCurve;
private Category[] rankedErrorVectors;

private Axis timeAxis;
private Axis xAxis;
private Axis yAxis;

/** The charts */
public XYChart progressChart;
public XYChart utilizationChart;
public VectorFieldChart vectorChart;

/**
 * Create a new Monitor
 */
public Monitor() {
    progress = new Series(2);
    utilization = new Series(2);
    rankedErrors = new VectorSeries[neighborhoodSize];

    timeAxis = new Axis(-1000, 0);  // display 1000 event packets
    timeAxis.setTitle("t");
    timeAxis.setUnit("ms");
    Axis ratio = new Axis(0, 1);
    ratio.setUnit("%");
    ratio.setTitle("");
    xAxis = new Axis();
    yAxis = new Axis();

    progressCurve = new Category(progress, new Axis[] {timeAxis, ratio});
    progressCurve.setColor(new float[] {0.0f, 0.0f, 1.0f});
    utilizationCurve = new Category(utilization, new Axis[] {timeAxis, ratio});
    utilizationCurve.setColor(new float[] {0.8f, 0.8f, 0.8f});
    rankedErrorVectors = new Category[rankedErrors.length];

    progressChart = new XYChart("Progress");
    progressChart.addCategory(progressCurve);
    utilizationChart = new XYChart("Utilization");
    utilizationChart.addCategory(utilizationCurve);
    vectorChart = new VectorFieldChart("Errors");
}

public void init() {
    // init statistics
    params = new Vector<String>();
    stat = new Vector<String>();
    utilizationStat = new ArrayList<Float>();
    startTime = System.currentTimeMillis();

    // remember sizeX and sizeY are unknown upon construction: set it now
    xAxis.setRange(-0.5, sizeX-0.5);
    yAxis.setRange(-0.5, sizeY-0.5);
    for (int i = 0; i < rankedErrors.length; i++) {
        rankedErrors[i] = new VectorSeries(sizeX, sizeY);
    }
    for (int i = 0; i < rankedErrors.length; i++) {
        rankedErrorVectors[i] = new Category(rankedErrors[i], new Axis[] {xAxis, yAxis});
        rankedErrorVectors[i].setColor(new float[] {1.0f, 0.0f, 0.0f});

    }
    for (Category c : rankedErrorVectors) {
        vectorChart.addCategory(c);
    }

    totalNeighbors = neighborhoodSize * (sizeX-1)*(sizeY-1);
    correctNeighbors = 0;
    for (int i = 0; i < neighbors.length; i++) {
        int last = neighborhoodSize;
        /* adapt neighborhood size at borders */
        int x = i / sizeY;
        int y = i % sizeY;
        if (x == 0 || x == sizeY-1) last = (last / 2) + 1;
        if (y == 0 || y == sizeY-1) last = (last / 2) + 1;
        /* check for accidental initial correct guesses */
        for (int rank = 0; rank < last; rank++) {
            if (adjacent(i, neighbors[i][rank])) correctNeighbors++;
        }
    }
    outstanding = 1.0f;

    displayTime = (int) (System.nanoTime() / 1000);
}

/**
 * Begin monitoring sequence.
 */
public void enter(int inputSize, int inputDuration) {
    this.inputDuration = inputDuration;
    referenceTime = (int) (System.nanoTime() / 1000);  // start timer...
```

```
        }

        /**
         * End monitoring sequence.
         */
        public void exit() {
            /* calculate parameters */
            int now = (int) (System.nanoTime() / 1000);
            float utilization = (float) (now - referenceTime) / (float) inputDuration;

            /* update display data */
            time += 1;
            this.utilization.add(time, utilization);
            this.progress.add(time, correctNeighbors / totalNeighbors);
            progressCurve.getDataTransformation()[12] = -time;  // hack: shift progress curve back
            utilizationCurve.getDataTransformation()[12] = -time;  // hack: shift utilization curve back

            /* update display if needed */
            if (now >= displayTime) {
                progressChart.display();
                utilizationChart.display();
                if (showFalseEdges) vectorChart.display();
                displayTime += 1000000 / DISPLAY_FREQUENCY;
            }

            /* check for current best value */
            if (correctNeighbors > currentBest) {
                currentBest = correctNeighbors;
                if (onResetWriteStatsAndExit) {
                    if (outstanding == 1.0f) {
                        params.add(""); // data set
                        params.add(learningWindowShape == 0 ? "Gaussian" : learningWindowShape == 1 ? "Triangle" : "Rectangle");
                        params.add(String.valueOf(learningWindowMean));
                        params.add(String.valueOf(learningWindowStandardDeviation));
                        params.add(""); // reinforcement
                        params.add(String.valueOf(neighborhoodSize));
                        params.add(inhibit2ndOrderNeighbors ? "On" : "Off");
                        outstanding /= 2;
                    } else if (1.0f - currentBest/totalNeighbors <= outstanding) {
                        stat.add(String.valueOf(current));
                        outstanding /= 2;
                    }
                    if (utilizationStat != null) utilizationStat.add(utilization);
                }
            }
        }

        /**
         * Write the stat data to 'topologyLearningStats.csv'.
         */
        public void writeStat() {
            if (params.isEmpty()) return;  // nothing to write
            try {
                String fieldSeparator = ",";
                BufferedWriter statFile = new BufferedWriter(new FileWriter("topologyLearningStats.csv", true));
                for (String param : params) statFile.write(param + fieldSeparator);
                statFile.write(100.0f - monitor.currentBest / monitor.totalNeighbors * 100.0f + "%");
                for (String date : stat) statFile.write(fieldSeparator + date);
                statFile.newLine();
                statFile.flush();
                if (utilizationStat != null) {
                    statFile = new BufferedWriter(new FileWriter("utilization.csv", false));
                    for (Float util : utilizationStat) {
                        statFile.write(String.valueOf(util));
                        statFile.newLine();
                    }
                    statFile.flush();
                }
                System.out.println(String.valueOf((System.currentTimeMillis() - startTime)) + " ms.");

                System.exit(0);
            } catch (IOException e) { throw new RuntimeException(); }
        }

        /**
         * Checks if two nodes are adjacent.
         */
        protected boolean adjacent(int i, int j) {
            if (i == j) return false;
            int dx = j / sizeY - i / sizeY;
            int dy = j % sizeY - i % sizeY;
            return dx*dx + dy*dy <= maxSquaredNeighborDistance;
        }

        /**
         * Event handler on changed neighbor.
         * Nodes at the border ignore inexistent higher ranked neighbors.
         */
        public void neighborChanged(int i, int newNeighbor, int[] neighbors, int rank) {
            assert newNeighbor == neighbors[rank];
            int last = neighborhoodSize;
//            /* adapt neighborhood size at borders */
//            int x = i / sizeY;
//            int y = i % sizeY;
//            if (x == 0 || x == sizeY-1) last = (last / 2) + 1;
//            if (y == 0 || y == sizeY-1) last = (last / 2) + 1;
```

```
//            if (rank >= last) return;
/* TODO: at the moment neighbors of border nodes do not have to be inside neigbors[0..last-1]
                                                        but only inside neighbors[0..neighborhoodSize-1] */
            /* check for changes in correctness */
            if (adjacent(i, newNeighbor)) {
                if (!adjacent(i, neighbors[last])) correctNeighbors++;
            } else if (adjacent(i, neighbors[last])) {
                if (!adjacent(i, newNeighbor)) correctNeighbors--;
            }
        }

        /**
         * Event handler on changed neighbor ranking.
         */
        private void neighborhoodChanged(int i, int[] neighbors) {
            int last = neighborhoodSize;
            int xi = i / sizeY;
            int yi = i % sizeY;
            if (xi == 0 || xi == sizeY-1) last = (last / 2) + 1;
            if (yi == 0 || yi == sizeY-1) last = (last / 2) + 1;
            for (int rank = 0; rank < last; rank++) {
                int j = neighbors[rank];
                int dx = j / sizeY - i / sizeY;
                int dy = j % sizeY - i % sizeY;
                if (dx*dx + dy*dy <= maxSquaredNeighborDistance) j = i; // hide correct edges
                // calculate (x,y) position of j
                int xj = j / sizeY;
                int yj = j % sizeY;
                // set vector target to j
                rankedErrors[rank].set(i, xj, yj);
            }
        }

    }

}
```