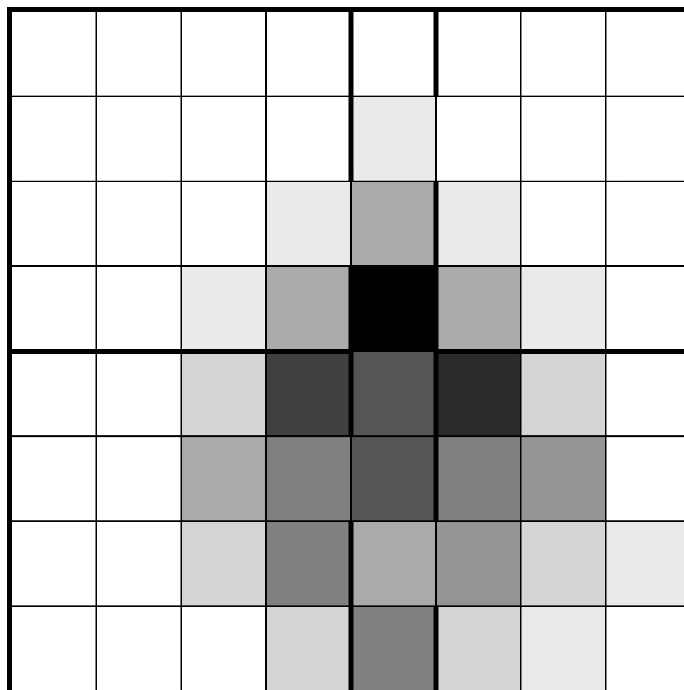
Towards a Sentient Environment Using a Neural Sensor Network

Freek van Polen



MASTER'S THESIS
COGNITIVE ARTIFICIAL INTELLIGENCE
UTRECHT UNIVERSITY

November 2, 2008

Towards a Sentient Environment Using a Neural Sensor Network

Freek van Polen

Master's Thesis
Cognitive Artificial Intelligence
Utrecht University

First Supervisor and Reviewer:
dr. Jurriaan van Diggelen
Utrecht University
jurriaan@cs.uu.nl

Second Supervisor and Reviewer:
dr. ir. Robbert-Jan Beun
Utrecht University
rj@cs.uu.nl

Third Supervisor:
drs. Jan-Peter Larsen
Almende BV, Rotterdam
jp@almende.com

Third Reviewer:
dr. Janneke van Lith
Utrecht University
Janneke.vanLith@phil.uu.nl

November 2, 2008

Contents

Preface **1**

1 Introduction **2**

1.1 Problem Statement 4
1.1.1 Scenario 4
1.1.2 Tasks 5
1.1.3 Aim 6
1.2 Design Decisions 6
1.2.1 Wireless Sensor Network 6
1.2.2 Binary Anonymous Sensors 7
1.3 Challenges 8
1.3.1 Energy Consumption 8
1.3.2 Distributed Application 9
1.3.3 Self-Organizing 9
1.3.4 Robustness 9
1.4 Approach 9
1.4.1 Performing Tasks 10
1.4.2 Neural Network 10
1.4.3 Contribution to the Field 10
1.5 Thesis Outline 11

2 Neural Networks **12**

2.1 Neurons in the Brain 12
2.1.1 Neurons 12
2.1.2 Plasticity in the Brain 13
2.2 Traditional Neural Networks 14
2.2.1 Architecture 14
2.2.2 Learning in Neural Networks 16
2.3 Spiking Neural Networks 18
2.3.1 Spike-Response Model 18
2.3.2 Learning in Spiking Neural Networks 19
2.4 Features of Neural Networks 21
2.4.1 Distributed 21
2.4.2 Learning 22
2.4.3 Robustness 22
2.4.4 Security of Data 22

3 The Tracking, Prediction and Identification Algorithm 24
3.1 Assumptions 24
3.2 Tracking 25
3.2.1 Related Work 25
3.2.2 Our Approach 26
3.2.3 Basis 27
3.2.4 Memory 28
3.2.5 Delay 29
3.3 Prediction 31
3.3.1 Related Work 31
3.3.2 Our Approach 32
3.3.3 Learning Procedure 33
3.4 Identification 35
3.4.1 Related Work 36
3.4.2 Our Approach 36
3.4.3 Introducing Identity 37
3.4.4 Propagating Identity 38
3.4.5 Learning Identity Specific Motion Models 39
3.5 Discussion 40

4 Validation using Simulation 41
4.1 Simulation Environment 41
4.2 Experimental Setup 43
4.2.1 Performance Measures 43
4.2.2 Parameter Settings 44
4.2.3 Experiments 45
4.3 Results 46
4.3.1 Tracking 46
4.3.2 Prediction 47
4.3.3 Identification 47
4.4 Discussion 48

5 Validation using Prototype 50
5.1 Prototype 50
5.1.1 Environment and Agent 50
5.1.2 Hardware 51
5.1.3 Sensor Network 52
5.2 Algorithm 53
5.2.1 Software Architecture 53
5.2.2 Parameter Settings 54
5.3 Results and Discussion 54

6 Discussion and Conclusion 58
6.1 Tasks 58
6.2 Challenges 59
6.3 Further Research 60
6.3.1 Algorithm Specific 60
6.3.2 Beyond the Algorithm 61
6.4 Conclusion 61

Bibliography **66**

Abstract

Ambient Intelligence is an emerging technology, and one domain where it can be applied is in intramural health care. In demented elders' homes, ambient intelligence could be used to track and identify people, so as to control environmental features, detect hazardous situations, make work more efficient, etc. We applied the paradigm of neural networks to sensor networks to create an algorithm that can track and identify agents in an environment. The algorithm is distributed, self-organizing and uses sensors that output only binary data. We tested the algorithm for its efficiency both in a simulation and for its feasibility in a real world prototype.

Preface

The answer to the question of what I am studying, Cognitive Artificial Intelligence, usually gets me puzzled looks and the question “How did you get the idea to do *that*?”. It makes me feel a bit embarrassed to have to say it was the movie “Terminator 3: The Rise of the Machines” that first gave me the idea. But after sitting through two hours of gunfire, exploding cars and imminent doomsday, I just had to know whether it was possible and if so, how soon. I decided on the spot to do a minor in Artificial Intelligence (I was studying Information Science at the time).

After I finished the bachelor in Information Science, I followed some more CAI courses for half a year, and then started a master in Cognitive Artificial Intelligence. I think one of the most appealing features of this master is that an attempt is made to model artificial intelligence with the human form of intelligence in mind. But also conversely, the secret behind human intelligence is sought to be unraveled by studying different models of artificial intelligence. Through this approach I have found myself ending up with an outspoken view on human intelligence, that one could term “anti-representationalist”. Unsurprisingly, it is my firm belief that if we aim at creating machines that resemble human intelligence, the traditional computational approach, where representations play an important role, should be abandoned.

I would like to thank Jurriaan van Diggelen and Robbert-Jan Beun, my supervisors, for being open-minded and letting me proceed with my very own project. Even though it meant applying the domain of spiking neural networks, which I had heard about but never studied, to the domain of wireless sensor networks, of which I knew possibly even less. It has been a fun experience, and very instructive both in terms of theoretical knowledge and in terms of the kind of hard work that is required to conduct valid and fruitful research. Many thanks also to Janneke van Lith, who agreed to be my third reviewer in the nick of time.

Furthermore, I would like to thank my girlfriend Renée for her unbridled confidence in me and everything I do, and for listening to me when I was excitedly blabbering about what I did, or when ranting about what didn’t work. I thank the people at Almende for giving me a good time and for creating an inspiring and supportive atmosphere. Also for providing a nice case from which to start my work and for facilitating the building of the prototype. Lastly I thank my parents for their unconditional faith in me taking the right decisions and for their support.

At the time of my graduation, work from this thesis has been presented and received with enthusiasm in both the ICIS (`http://www.icis.decis.nl`) and the ALwEN (`https://www.alwen.nl`) projects.

Chapter 1

Introduction

One of the major political concerns in the Netherlands today is the aging population. An ever larger percentage of the population is above the age of 65 years (see Figure 1.1), through both baby-booming after the second World War, an increasing life expectation, and low birth rates. This phenomenon can be observed not only in the Netherlands, but in countries of the European Union and the Organization for Economic Cooperation and Development in general [1]. This phenomenon naturally increases the demand for health care and thus induces the need for the availability of health care to grow proportionally. Moreover, it poses a heavier tax burden on the working portion of the population, as higher expenditures on pension and health care must be financed by a smaller portion of the population [2]. Thus, the availability of health care should not only grow, it should should also cost less. This can be achieved by for instance making work more efficient, and creating a better match between the demand and the availability of health care.

One way to reduce costs in elderly care is ambient intelligence technology [3, 4, 5]. Ambient intelligence technology is about outfitting an environment with many computer nodes, that are capable of sensing and acting on their environment. These computer nodes gather data from the environment using their sensors; devices that measure some value in the environment, such as cameras, microphones, thermometers, etc. The computer nodes will process the data they gather and then act upon the environment using actuators, or present information to human beings for which this information is relevant. In [4] a number of scenario's are presented to illustrate the forms ambient intelligence could take on. These forms range from fridges that are aware of their contents to cars that can determine the smoking behavior of their drivers to devices that can be worn in clothes or around the wrist that take care of the communication or daily schedules of their user.

Most work on ambient intelligence for elderly care focuses on a decentralized form of care, where care is delivered to people in their own home, as opposed to institutionalized elderly care, such as (demented) elderly homes. This is because of two reasons. First, institutionalized elderly care is usually a lot more expensive than care delivered to people in their own home, and thus it is desirable to keep people at home as long as possible [6]. Second, allowing people to live independently significantly increases patient happiness [7, 8]. Ambient intelligence can help in elderly care at home in the form of for instance Body-Area

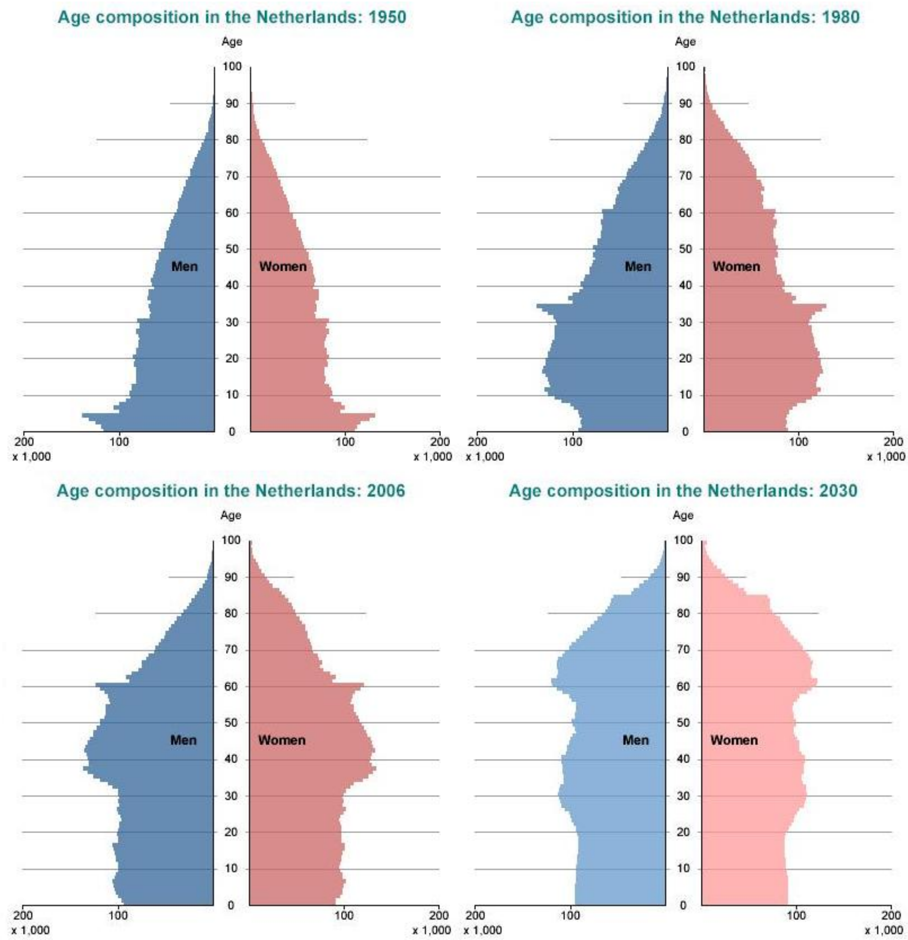


Figure 1.1: The age composition of the Dutch population in the years 1950, 1980, 2006, and a forecast for the year 2030. Source: Dutch Central Bureau for Statistics (CBS, `http://www.cbs.nl`).

Networks, where sensors, worn by a patient, constantly monitor the patient's vital signs, and sound an alarm if anything is wrong [8]. Another application of ambient intelligence is to equip the patient's house with sensors to monitor and assist a patient in the performing of several activities [7].

That a decentralized form of elderly care is to be preferred over institutionalized elderly care results in a diminishing demand for institutionalized elderly care. It is, however, questionable whether the former can completely replace the latter [9]. Thus, reducing the costs of institutionalized health care has been identified as a potential application of ambient intelligence [5]. In this thesis, we will use a specific scenario to construct a preliminary proposal for an ambient intelligence system for institutionalized elderly care. However, we believe our results to be useful in other areas too.

1.1 Problem Statement

In this section we will present the aim of this thesis. We will use a scenario to illustrate what use an ambient intelligence system might have in institutionalized elderly care. Based on this scenario we shall define the tasks the system will have to fulfill. We will then state the aim of the thesis.

1.1.1 Scenario

The Humanitas Foundation (`http://www.humanitas.nu`) is a health and elderly care organization located in the city of Rotterdam in the Netherlands. Humanitas continually strives to improve the health care experience of its customers. For instance by allowing them to live autonomously as long as possible and tailoring the application of care to the specific needs of every customer. The Hannie Dekhuijzen demented elderly home is one of Humanitas' clinics in Rotterdam. It is interested in ambient intelligence to make work more efficient and to create a better match between demand and availability of health care. In cooperation with Almende BV (`http://www.almende.com`), an IT-research company in Rotterdam, some specific scenarios have been constructed where an ambient intelligence system could be useful. [10]

People counting

In cases of emergencies, it is highly desirable to be aware of the number of people present in a building, and their locations in the building. Especially in a demented elderly home, occupants typically are not capable of escorting themselves to safety if, for instance, a fire breaks out. It is then important for both employees and firefighters to know how many people are in the building and where they are.

Geofencing

The inhabitants of demented elderly home typically are not supposed to be able to walk around the entire building. Since inhabitants do not always know where they are, where they are going and how to get back, it is much more efficient to keep them where they should be then having to escort them back every now and again. An ambient intelligence system could do this by, for

instance, controlling a door and only opening it for people that are allowed to pass through. Alternatively, the system might also help to send elders that have become lost back to their room by using, for instance, lights in the floor to show an elder the way.

Controlling environment

The inhabitants of a demented elderly home will typically not be able to properly control some environmental features such as heating, light, elevators etc. Though nurses could take care of these issues, it would be more efficient if they would be regulated automatically. An ambient intelligence system could do this by for instance automatically switching on and off light and/or heating when someone walks in and out of a room or sending elevators to floors where someone might want to use them.

Security

Because a demented elderly home should be open to visitors, it is generally quite easy for burglars to gain access to the building. If an ambient intelligence system could recognize the distinct behavior that a burglar would typically exhibit (i.e. walking up and down hallways trying doors), it could alert employees of a burglar's presence, and thereby enhance security.

Toilet Management

It appears that demented elders sometime fail to notice they need to use the toilet, and instead get some inane feeling that makes them wander up and down a hallway. It is obvious that it is more efficient to catch these elders while they are wandering up and down a hallway than when it's too late. If an ambient intelligence system would be able to recognize this pattern of walking, it could alert a nurse and the situation could be resolved.

1.1.2 Tasks

From the scenarios we can extract a number of information gathering tasks that the system would need to perform. First, each of these scenarios demands that the network is able to determine the *location* of agents in the environment. More specifically, every computer node that is part of the system, must at all times know whether an agent is at or near the node's location in the environment or not. This is a common task for ambient intelligence systems, see for instance [11, 12], and is often referred to as *tracking*.

Though for the first scenario the system only has to be able to track agents in the environment, the other scenarios demand some more intricate information. For controlling the environment and geofencing, it is desirable that the system is able to *predict* the location of agents in the future. More specifically, every computer node that is part of the system must know an agent is coming to its location before the agent is actually there. If the system is able to do this, it can send elevators to floors in time, make sure the lights in a room are on before someone enters, close a door before a patient walks through, etc. We will call this task *prediction*.

Third, a number of these tasks require that the system can also determine the *identity* of the agents in the environment. The precision with which one would want to identify agents differs per scenario. For the second scenario one might want to have a unique reference for every agent in the environment, so that one can define an area in which an agent is allowed to move freely for every individual agent. For other scenarios it might be enough if the system is merely able to distinguish between patients, nurses and burglars, allowing an identification in classes of agents. For now we will assume that we are interested in classes of agents. Thus, we define the *identification* task as follows: every node that is part of the system and that has determined an agent is near its location in the environment, must also be able to determine the class of that agent.

1.1.3 Aim

We are now ready to define the aim of this thesis.

> *In this thesis we will develop an elementary ambient intelligence system that is capable of performing the tasks: tracking, prediction and identification.*

Thus, though ambient intelligence systems are defined to sense their environment, process the raw data that is gathered and act upon their environment, we will limit ourselves to the sensing and processing. This leaves the question open of how to use the information gathered by the system in a useful way, as the ultimate goal is to develop a system that may assist or even substitute the personnel of a (demented) elderly home.

1.2 Design Decisions

Any system consisting of computer nodes that sense, reason about and act upon their environment may be called an ambient intelligence system, but of course these exist in many variations. One major distinction that can be made is between single sensor systems and sensor networks, where data from multiple, and possibly different sorts of sensors is fused. The tasks our system has to perform clearly demand that multiple sensors are spread throughout the environment, and so we must use a sensor network. Note that since we do not focus on the acting capabilities of the system, we shall from now on speak of our system as a sensor network.

1.2.1 Wireless Sensor Network

Sensor networks can be wired or wireless. For nodes in a wired network energy in principle is available without limit and communication bandwidth is not a limiting factor. Nodes in a wireless sensor network are battery operated and employ short range wireless communication. Nodes in a wireless sensor network are designed for swift and unorganized deployment in an environment where the infrastructure needed for a wired sensor network is not present, or where the installation of such an infrastructure is not feasible. Environments where this

is the case are for instance existing factories or plants, forests, hostile territory in military operations. We will assume a *wireless sensor network*.

Using a wireless sensor network instead of a wired sensor network brings with it some distinct advantages concerning the *costs of the network*. Since the reason for designing this sensor network is to reduce costs of institutionalized elderly care, costs are a significant factor.

First, *no infrastructure is needed* to deploy a wireless sensor network. Every node carries its own power supply, and no cables are needed to facilitate the communication between nodes. In some environments, such as a forest, this is not so much an advantage as it is a requirement. In other environments, like buildings or factories it will depend on the size of the building, the infrastructure already present or the costs of installing the infrastructure, how big an advantage this is.

Second, research is ongoing to decrease the *costs of individual wireless network nodes*. This research is stimulated by the fact that wireless sensor networks are typically being used in large environments, thus allowing for great potential savings if individual nodes are cheap. Also, environments where installing infrastructure is impossible, such as in a forest, typically also are environments where individual nodes have a high probability of breaking down, or never being retrieved.

Two projects in which work is being performed to decrease the cost of wireless sensor network nodes are the ALwEN project and the quest for *smart dust*. The ALwEN project (`http://www.alwen.nl`), which started in early 2008, aims at creating a platform for ambient intelligence. As the aim is at creating wireless sensor networks consisting of up to 10000 nodes total, one goal is to create nodes that cost not more than a few euro's[13].

Research is even ongoing to create *smart dust*; wireless sensor nodes the size of a speck of dust and costing not more then a few cents [14, 15]. The nodes may be scattered in massive amounts in an environment, and start collecting data. The goal is to create wireless sensor networks that can be deployed in an environment randomly and that can deal with catastrophic losses of nodes.

1.2.2 Binary Anonymous Sensors

With the decision of using a wireless sensor network in mind, we have to decide what kind of sensors we will use. Ideally, to track agents in an environment, one would outfit every agent with a node with Global Positioning System (GPS) functionality that records the location of the agent at all times. This would also instantly solve the identification task, since it would then be trivial to add an id to every node. However, the argument of costs effectively prohibits the use of nodes with GPS functionality altogether. Moreover, GPS does not work indoors.

A more affordable option would be the use of RFID (Radio Frequency Identification) tags: small devices that can be attached to objects and that carry their own unique id [16]. These id's can be read by an RFID reader if it is in a range of, depending on the kind of tag, 30 cm to some meters. Prices of RFID tags are negligible, making tagging agents with RFID tags and outfitting the environment with readers viable. This would make tracking and identifying agents in the environment a trivial exercise again. However, other problems arise, the main one being privacy concerns. First, RFID technology is not secure enough

yet to protect RFID tags from theft of their unique id [16]. If we associate the identity of agents with an RFID id, the system will become vulnerable to identity theft, which has been identified as a major threat for ambient intelligence systems [17]. Second, we can not rely on elderly people to always carry their RFID tag with them if it is integrated in some sort of card. We could solve this by applying the RFID tags to such objects as shoes, canes, wheelchairs etc [7], but this would make identity theft even more trivial [17]. A burglar would but need to grab a cane from a demented elderly person and then be free to move around.

Instead, we will limit ourselves to nodes that are at a *fixed location* in the environment, outfitted with sensors that are *binary* and *anonymous*. Binary and anonymous sensors are sensors that can output only one bit of data per reading, such that from one single reading one can never directly determine the identity of an agent in the environment. Examples of this sort of sensors are motion detectors, breakbeam sensors, which detect whether a beam has been interrupted, pressure mats, contact switches, which can measure if a cabinet is open or closed, etc. This constraint will make performing the tasks that we are facing much more difficult. This is because we have the absolute minimum of information (at one sensor reading per second, only one bit per sensor per second, compared to for instance an image per second for a camera). But there are good arguments for imposing these constraints on the network.

First, research has shown that elders would feel sensors that can determine their identity to be a serious violation of their *privacy* [7]. Thus, using such sensors would endanger the acceptation of the system. Second, because we will be using nodes that are limited in both their computational and their communication capabilities, we will not be able to process the amount of data that sensors like cameras or microphones can gather. The bandwidth is not available to send the data to some processing unit, and the nodes themselves are not capable of processing so much data. Third, if we are able to design a system that performs the above mentioned tasks using only binary and anonymous sensors, the performance of the system on these tasks can only improve if more complex sensors are added. Thus, we will never be in a position where the system relies on the use of complex sensors for its performance, and will thus be robust to the loss of such sensors for whatever reason.

1.3 Challenges

The design decisions we have made pose a number of challenges and demands to the system.

1.3.1 Energy Consumption

The first challenge is that since our individual nodes will be powered by batteries, energy consumption becomes an issue that has to be taken into account. Specifically, it is widely acknowledged that for wireless sensor nodes communication is more energy consuming than computation [11, 12, 18, 19, 20], and so we will have to take care to create a balance between the two that will make sensor nodes last as long as possible.

1.3.2 Distributed Application

The second challenge is that applications for wireless sensor networks of the size we are aiming at need to be distributed. This means that every node in the network takes care of a small part of he computation involved in the application. Moreover, nodes do so using only local data: data gathered either by itself or other nodes that are in the vicinity. The reason that using a centralized approach, where data is sent to a central processor, can not be used is that the communication capabilities of nodes are too limited. Nodes will typically not have a communication range that enables them to communicate with a central station directly. Thus, a centralized approach will demand individual nodes to double as router. Especially nodes that are close to a central station will be heavily burdened by this [21]. Since communication bandwidth is limited, the routing task will quickly become too heavy for individual nodes. Moreover, because communication consumes a lot of power, it is undesirable to burden nodes with a lot of communication. Instead, creating a distributed application will limit communication to a local scale.

1.3.3 Self-Organizing

In a wireless sensor network consisting of thousands of nodes, it is not feasible to program every node individually with data specific to that node. Data specific to a node include data concerning the location of a node in the environment that is required for performing tasks, or data on which other nodes the node can communicate with. Rather, nodes should be programmed in a uniform way and, once deployed, organize themselves into a working network. This means nodes will have to find and identify neighboring nodes themselves, and somehow learn any data that they require for the application.

1.3.4 Robustness

The last challenge we identify is one of robustness. A network of cheap and small nodes may not be as reliable as a more expensive, wired sensor network. Nodes may be prone to breaking down or getting lost. Moreover, the wireless communication facilities of wireless nodes may cause a high rate of delayed or missing messages. Also, the choice of using binary and anonymous sensors leaves us with little possibilities of checking the reliability of individual sensor readings. Thus, the network will likely be faced with an amount of noise in the input, delayed or missing messages, and broken nodes. Any application developed for a wireless sensor network must be robust to these issues, and not suffer too much from them in terms of performance.

1.4 Approach

We have made a number of design decision, and identified the challenges that these decisions pose to our system. It is now time to discuss the approach we will take to solve the tasks that we have set for the system, while dealing with the challenges that we identified.

1.4.1 Performing Tasks

Our system will have to operate distributed, and thus every node will have to take part in performing the tasks, using only local data. For the *tracking* task, every node has to determine whether an agent is present at its physical location in the environment. To do this, every node will use the data its own sensor gathers, and data that sensors on nodes in the vicinity gather. Nodes will not only use the most current sensor data, but also data from the past.

For the *prediction* task, every node has to determine whether an agent is coming to its location or not. To do this, nodes will attempt to learn the typical behavior of agents. Using the typical behavior that agents exhibit and current sensor readings, nodes can determine whether an agent is coming or not.

For the *identification* task, nodes have to be able to determine the class of an agent that they have determined to be at their location in the environment. With the absence of sensors that can directly identify agents, some other means will have to be employed. We will let nodes learn the typical behavior of individual classes of agents. By comparing the behavior an agent is currently exhibiting with the typical behavior of different classes of agents, the class of an agent can be determined.

1.4.2 Neural Network

We have to create an application that is distributed, self-organizing, robust to noise and, moreover, able to perform the tasks the way we described above. Thus, the application must allow individual nodes to learn typical behaviors of agents. These requirements inspire us to adopt the paradigm of artificial neural networks. More specifically, we will view the entire wireless sensor network as one artificial neural network. Artificial neural networks are distributed and robust to noise and breaking down of individual network components by their very nature. We will design the neural in such a way that it is fully self-organizing. Last, neural networks are capable of learning, and thus adopting neural networks provides us with a means to learn the typical behavior of agents.

1.4.3 Contribution to the Field

The novelty of this work lies in two issues. First, this is, to the best of our knowledge, the first attempt to create a wireless sensor network that can perform the three tasks we defined, while being distributed, self-organizing, robust to noise and using only binary and anonymous sensors. Though much work has been done on performing each of these three tasks using wireless sensor networks, not one work has been found that takes each of these challenges into account.

Secondly, to the best of our knowledge, this is the first attempt at applying the paradigm of artificial neural networks to wireless sensor networks. Moreover, the attempt to create an actual physical neural network, rather than a computer simulated neural network is rather new. It is, however, our belief that the field of wireless sensor networks is especially suitable for creating physical neural networks.

1.5 Thesis Outline

In the next chapter we will give a brief introduction to the field of neural networks. We will first introduce what Maass [22] calls the first and second generation of neural networks, referred to as traditional neural networks. We will then discuss what he calls the third generation of neural networks; spiking neural networks. This chapter serves both to introduce terminology and to argue why the paradigm of neural networks is fit for our needs.

In chapter 3 we will present the algorithm that will operate on a wireless sensor network and is capable of performing the three tasks mentioned above: tracking, prediction and identification. For each task, we will first review work that has been done on performing that task using wireless sensor networks, before presenting our own approach.

In chapter 4 we will perform an analysis of the performance of the algorithm on the three tasks. We will use a simulation to test the algorithm on the three tasks both in noise-free and noisy scenarios. We will present a number of relationships between different parameters of the algorithm and the performance and further discuss the results.

In chapter 5 we shall present a partial implementation of our algorithm in a real world prototype. We will use this prototype to analyze the practical feasibility of the algorithm when implemented on limited computational devices such as the wireless sensor nodes we intend to use.

In the sixth chapter we shall discuss the algorithm, based on the validation by simulation and the validation by prototype. We will first discuss whether the algorithm we proposed performed the tasks to a satisfying extend, and whether it deals with the challenges we have identified. We then will give various directions for further research and conclude the thesis.

Chapter 2

Neural Networks

Artificial neural networks (henceforth simply neural networks or nn's), are a computational paradigm that is inspired by the functioning of the human brain. The brain consists of billions of neurons; electrically excitable cells that are interconnected in a myriad of ways. A neural network is essentially an attempt to create a small brain-like structure consisting of artificial neurons. Though neural networks have been around since the 1940's, it was not until the mid 80's that they received any real attention. Since then, neural networks in many variations have emerged, and they have been used for many different purposes. The different variations of neural networks can be divided into three major generations [22], which we will review in turn. However, we will first briefly review the workings of neurons in the brain, so as to better understand the working of neural networks. We will end the chapter by discussing several features of the paradigm of neural networks and argue why they are fit for our needs.

2.1 Neurons in the Brain

An extensive description of the exact working of single neurons is outside the scope of this thesis, and we will here review only those aspects that are necessary to understand the rest of the chapter. We refer the interested reader to [23] for an elementary introduction to neuroscience, and to [24] for an introduction to neuron models from a computational point of view. Furthermore we would like to encourage the interested reader to search, for instance, on the web, as a wealth of material on neuroscience exists.

2.1.1 Neurons

A typical neuron looks roughly as illustrated in Figure 2.1. In the cell body an action or membrane potential is built up under the influence of incoming electrical pulses. These pulses are received through the dendrites, and come from other neurons. Depending on the type of the neuron sending the pulse, the pulse will cause the action potential to either rise or fall. If the action potential of a neuron deviates from its so called resting potential, for instance through a number of incoming pulses, it will gradually converge back to this resting

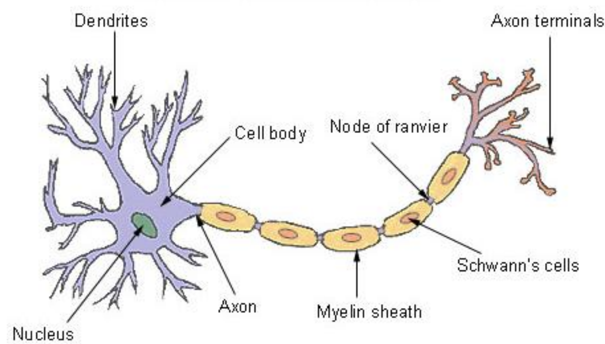


Figure 2.1: The typical structure of a neuron in the brain.

potential. One should note that variations in action potential of a neuron take place in the millisecond scale. A neuron that has been heavily excited will have its potential converge back to the resting potential in tens of milliseconds.

Once the action potential of the neuron crosses some threshold from below, it discharges, and we say it *fires* or *spikes*. Note that for the action potential of a neuron to cross this threshold, generally a burst of incoming pulses is required. Upon discharging, the action potential of the neuron falls steeply to a very low value, effectively creating a *refractoring* period. During this period, which lasts some miliseconds, the neuron can not spike, no matter the amount of pulses received from other neurons. Over time, the potential automatically converges back to the resting potential.

When a neuron discharges, it sends out an electrical pulse of its own. This pulse travels down the axon, and reaches the axon terminals. At the axon terminals there is a small space, where the axon terminal of this neuron and a dendrite of another neuron join. This space is called a synapse. When the electrical pulse reaches the synapse, the axon terminal will release an amount of neuro-transmitter chemicals. These chemicals traverse the synapse, and cause an electrical pulse at the receiving neurons dendrite, thus completing the journey of the electrical pulse to other neurons.

In the human brain, neurons of many different sorts exist. We already made the distinction between neurons that excite the action potential of other neurons and neurons that inhibit the action potential of other neurons. Other variations exist for instance in the firing regime (e.g. a pattern of spikes over time) neurons settle in when fed a constant stream of incoming pulses. Since the brain consists of billions of different types of neurons, connected by many kilometers of axons and dendrites, the exact way in which these neurons interact to create human intelligence is as of yet unknown.

2.1.2 Plasticity in the Brain

An important feature of the brain is that it is not static, but plastic. The efficiency with which a neuron is capable of eliciting a reaction from another neuron by sending pulses may be subject to change over time. This phenomenon is termed synaptic plasticity, and exists for both the long and the short term.

Short term plasticity exists on the millisecond scale, for instance during the refractoring period of a neuron, when incoming pulses will have little to no impact on the action potential of the neuron. Evidence also exists that if a neuron sends a burst of pulses, the first of these will have more impact than the last [24].

Synaptic plasticity for the long term induces change that lasts for hours, days or even longer. Hebb [25] already conjectured in 1949 that some form of synaptic plasticity should be present in the brain. He stated that if a neuron A persistently had a part in the firing of neuron B, neuron A would grow more efficient in firing neuron B. This way clusters of simultaneously firing neurons would evolve. From empirical evidence, it indeed appears that such synaptic plasticity exists in the brain [26]. It is important to note that, even though Hebb's postulate says nothing of neurons growing less efficient, this certainly does take place in the brain. Long term synaptic plasticity is often referred to as Long Term Potentiation (LTP) and Long Term Depression (LTD). We shall not delve further into synaptic plasticity in the actual brain, but merely stress the point that it exists. Later, when discussing artificial neural networks we shall return to this point.

The brain is plastic in another way. When a neuron spikes, the electrical pulse it sends out naturally takes some time to journey from cell body to cell body. Empirical evidence suggests that the precise timing of spikes and thereby the time pulses reach a next neuron, are of importance in for instance the auditory system of the barn owl [24]. It appears that the delay between the spiking of a neuron, and the arrival of the pulse at the next neuron may be increased or decreased to, for instance, achieve more synchronicity between the firing times of neurons. The delay may be changed by for instance changing the length or thickness of an axon [27].

2.2 Traditional Neural Networks

Traditional neural networks (for the rest of this section, we shall omit the "traditional"), though inspired by the brain, do not resemble it very much. Neural networks consist of neurons, as the brain, and the efficiency of connections between neurons may change. This last feature is called *learning*. However, artificial neurons are not nearly as complex in their behavior as biological neurons, and the same goes for the learning procedure of neural networks compared to plasticity in the brain. As a computational paradigm neural networks do have their merits, however. We will first discuss the basic architecture of neural networks, then outline the various learning procedures and finally. As an extensive discussion of the field of artificial neural networks is not within the scope of this thesis, we refer the interested reader to [28].

2.2.1 Architecture

The neural network consists of a number of neurons, connected to form a specific architecture. Every neuron in the network has a variable *activation*, analogous to the membrane potential of neurons in the brain. Associated with every connection between two neurons in the network is a *weight*, which represents the efficiency of the connection. The typical neural network is arranged in

either two or three layers: an *input layer*, possibly a *hidden layer*, and an *output layer*. Connections exist between neurons of subsequent layers, but not between neurons of the same layer or between neurons that are more than one layer apart (see Figure 2.2). The input layer is where input enters the network. The hidden layer, if present, is called this way because it is somewhat “hidden” from the outside world. The output layer is where the network generates output.

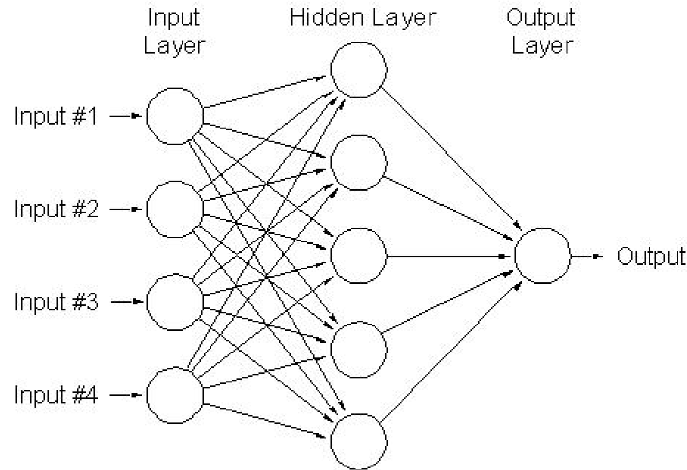


Figure 2.2: The general architecture of a neural network consisting of an input layer, a hidden layer and an output layer.

To operate the network, one first sets the activation of neurons in the input layer to some value. Then subsequent layers of neurons use the activations of neurons in the preceding layer and the weights of connections between the two layers to compute their own activation. This process goes on until the neurons in the output layer have computed their activation, and the network produces output. Formally, the activation A of neuron i is some function f of u, where u is the sum of the activations of all neurons j in the preceding layer that are connected to i, multiplied by the weight w between i and j.

$$u_i = \sum_j w_{ij} * A_j \tag{2.1a}$$

$$A_i = f(u_i) \tag{2.1b}$$

The first generation of neural networks employed a boolean function for f (see 2.2a), and are called perceptrons. Two layered perceptrons (i.e. consisting only of an input layer and an output layer) are able only to solve problems that are linearly separable, which means they are unable to solve for instance the XOR problem [29]. In logic, p XOR q means that either p is true or q is true, but not both. If we draw a table with the truth values of p and q, as in table 2.1, we can see that the two cases where p XOR q is true can not be separated from the two cases where it is not by a single linear line. It was later proven that multi-layer perceptrons (i.e. consisting also of a hidden layer) were able to solve non-linearly separable problems.

q \ p	0	1
0	0	1
1	1	0

Table 2.1: The XOR problem is an example of a problem that is not linearly separable.

The second generation of neural networks employs linear (see 2.2b) or sigmoidal functions (see 2.2c) for f, and is considerably more powerful [22].

$$f(u_i) = \begin{cases} 1 & \text{if } u_i \geq \theta \\ 0 & \text{otherwise} \end{cases} \tag{2.2a}$$

$$f(u_i) = u_i \tag{2.2b}$$

$$f(u_i) = \frac{1}{1 + e^{-u_i}} \tag{2.2c}$$

Moreover, activation functions that allow neurons to have analog instead of digital activations are biologically more realistic. In the brain computation takes place continuously rather then in batches, as in neural networks. Thus, the batches in which neural networks operate can be seen as a discretization of the continuous timescale at which real neurons operate. In neural networks of the first generation, the activation of a neuron can only be 1 or 0, meaning one can only represent whether the neuron fired or not this time step, or whether it fired more than some threshold. Neural networks of the second generation allow one to model the real firing rate of a neuron during a time step.

2.2.2 Learning in Neural Networks

We discussed earlier perceptrons solving non-linearly separable problems. The propagation of activations of neurons through the network completely depends on the values of the weights of the connections. Thus, neural networks are only capable of solving problems if the weights in the network are assigned the correct values. The most appealing feature of neural networks is that they are able to learn these weight values. Typically, the weights of the connections in a network are initialized to random values, causing the network to output random values. However, by applying a learning procedure after every time an input is presented, over time the weights of the network can be automatically adjusted such that the network will perform better.

Presenting an input to the network, gathering the output and applying the learning procedure is called an *iteration*. Every iteration, the learning procedure updates every weight with some Δw_{ij}. The learning procedure that is used determines how this Δw_{ij} is calculated. We will give here two examples of learning procedures, one supervised, the other unsupervised. Though these two examples by no means cover the entire spectrum of learning procedures available for neural networks, they will give a feeling for what is possible.

Supervised Learning

Supervised learning procedures require a teacher, external to the neural network, to tell the network what the correct output for a certain input was. Thus, to train a network in a supervised way, one needs a training set, consisting of pairs of input data and correct output data. Possibly the most famous supervised learning procedure is *backpropagation* [28, 30]. Suppose we have a network consisting of a number of input neurons j and a number of output neurons i, all employing a linear activation function as in 2.2b, and no hidden neurons. Because we know for each input the desired output T for every output neuron, we can define an error measure E on the network for any particular input:

$$E = \frac{1}{2} \sum_i (T_i - A_i)^2 = \frac{1}{2} \sum_i (T_i - \sum_j w_{ij} * A_j)^2 \tag{2.3}$$

Note that the better the network performs, the smaller the error E for inputs is. By differentiating E towards some w_{ij}, we can see if we have to make w_{ij} smaller or larger to make E smaller. Thus, we can use the following update function:

$$\Delta w_{ij} = -\alpha * \frac{dE}{dw_{ij}} = \alpha * (T_i - A_i) * A_j \tag{2.4}$$

Here the $-$ makes sure the network learns in the correct direction, i.e. towards a smaller error. The factor α is often called the *learning rate*, and determines how fast the network learns. This technique of determining the weight change for every weight is called *gradient descent*. The nice thing is that if we have multiple layers, we can simply expand the error function E and differentiate it also towards weights that are buried deeper in the expression. This learning procedure is typically repeated for some time, using inputs in the training set multiple times, until the error of the network on the entire training set is below some fixed threshold.

Unsupervised Learning

In some cases, a training set is not available because the correct output for inputs are not known. In this case, unsupervised learning procedures exist that are capable of creating clusters in the set of inputs. Where a supervised learning procedure may be used to for instance teach a network to classify a painting as either a Van Gogh or not a Van Gogh, an unsupervised procedure will create a clustering in the set of paintings it has seen, such that hopefully one cluster will correspond to Van Gogh paintings, while the other corresponds to paintings that are not Van Gogh's.

One well known neural network that uses unsupervised learning is the Kohonen Network [31]. The Kohonen network consists only of an input layer and an output layer, and an alternative activation function is used for the output neurons (see 2.5a). Every iteration, only the output neuron with the lowest activation and some of the neurons that are nearby in the output layer get to learn. The weights are updated towards the activation of the input neurons (see 2.5b), such that the next time this or a similar input is presented, this neuron will do even better. This way, output neurons that are near one another in the output layer will learn to respond to similar inputs, and thus a clustering arises.

$$A_i = \sqrt{\sum_j (A_j - w_{ij})^2} \tag{2.5a}$$

$$\Delta w_{ij} = \alpha * (A_j - w_{ij}) \tag{2.5b}$$

2.3 Spiking Neural Networks

We mentioned earlier that neural networks from the second generation are biologically more plausible than neural networks from the first generation. Both types of network discretize the continuous time that real neurons operate in. However, networks of the first generation can only represent whether a neuron spikes or not during an interval, while networks from the second generation represent the firing rate of a neuron during an interval. Thus, more information can be encoded by a single neuron. Over the years, however, empirical evidence has gathered that suggests that real neurons encode information not only by their average firing rate, but also by the precise timing of individual spikes [22, 24, 32, 33].

The third generation of artificial neural networks consist of *spiking neurons*, and treat the timing of individual spikes as the carrier of meaning instead of rates of spikes. This shift has some major consequences. To calculate the activation of a neuron, one will have to take into account the individual spikes of connected neurons. Also, by considering the timing of individual spikes to be the carrier of meaning, we have introduced time as a factor. It will no longer suffice to have the network operate in batches that are not related to one another. Rather, spiking neural networks have to operate on a time scale.

2.3.1 Spike-Response Model

This addition of time as a factor calls for different activation functions for individual neurons. There are two notationally different models for the activation of spiking neurons: *Integrate-and-Fire* models and *Spike-Response* models. The first makes use of differential equations, the second of kernels that account for different features in the model. Both models are functionally the same, and both models exist in variations of complexity. The most complex models are aimed at mimicking the exact dynamics of individual neurons in the brain, and are thus unfit to simulate populations or networks of neurons. The more simple models are less expressive at the level of individual neurons but are computationally less expensive and so more fit for simulating networks of spiking neurons[1]. We will only describe the most general Spike-Response model, as the algorithm we will propose borrows mainly from Spike-Response models. For an extensive introduction to spiking neural models of different complexity, and the Integrate-and-Fire model, we refer to [24].

There are three phenomena of influence on the activation of a spiking neuron: a refractoring period after the neuron has spiked, spikes of connected neurons and external input. Each of these three phenomena are handled independently

[1]Izhikevich [32], however, suggests an Integrate-and-Fire model with only four parameters that is both computationally feasible, and can model at least twenty behaviorally different types of neurons that are found in the brain.

by three different kernels, and thus, the activation function for a neuron i consists of three different parts. The η kernel models the refractoring period a neuron i experiences after it has spiked. Thus, this kernel depends on the current time t and the time of the last spike of the neuron $\widehat{t_i}$, see equation 2.6a. A neuron i spikes if its activation A reaches a certain threshold ϑ from below, and upon spiking the time of the last spike of the neuron $\widehat{t_i}$ is updated. The ϵ kernel models the response of the activation of neuron i to individual incoming spikes from connected neurons j. The effect of an individual spike f on the activation of a neuron decays over time, and hence the kernel depends on the difference between the current time t and the time at which spike f from neuron j arrived $t_j^{(f)}$. The maximal effect of the spike on the activation of neuron i also depends on the last time neuron i itself spiked, since during the refractoring period neurons are effected less by incoming spikes. Since every spike of every connected neuron in principle has an everlasting effect, and the effect of spikes is influenced by the weight of a connection, we sum over the weights of all connections and over all past spikes. See equation 2.6b. The third kernel is the κ kernel, which models the neuron's response to an external input current I^{ext}. The kernel has the same dependencies as the ϵ kernel, but since the input current is continuous instead of discrete like incoming spikes, we have to use an integral instead of a summation. See equation 2.6c.

$$A_i(t) = \eta(t - \widehat{t_i}) + \tag{2.6a}$$

$$\sum_j w_{ij} \sum_f \epsilon(t - \widehat{t_i}, t - t_j^{(f)}) + \tag{2.6b}$$

$$\int_0^\infty \kappa(t - \widehat{t_i}, s) I^{ext}(t - s) \mathrm{d}s \tag{2.6c}$$

Typical η kernels will generate a large negative value right after a spike, which then decays back to a value of 0 over time. An example is equation 2.7a, where τ_I and τ_R are parameters that determine the shape of the function. The function is drawn in Figure 2.3(a) for $\tau_I = -1$ and $\tau_R = 4$. A typical ϵ kernel will let incoming spikes first increase the activation of a neuron swiftly, after which the effect decays again over time. A possible ϵ kernel that is stripped of its dependency on the last spike of neuron i (the shape of the effect of incoming spikes is always the same) is equation 2.7b. τ_M and τ_S are again parameters that allow one to change the shape of the function. The function is drawn in Figure 2.3(b) for $\tau_M = 4$ and $\tau_S = 1$.

$$\eta(t - \widehat{t_i}) = -\tau_I * \frac{-(t - \widehat{t_i})}{\tau_R} \tag{2.7a}$$

$$\epsilon(t - t_j^{(f)}) = exp(\frac{-(t - t_j^{(f)})}{\tau_M}) - exp(\frac{-(t - t_j^{(f)})}{\tau_S}) \tag{2.7b}$$

2.3.2 Learning in Spiking Neural Networks

Learning has a prominent role in traditional neural networks, as these networks must go through a learning phase to be able to perform any task. For spiking

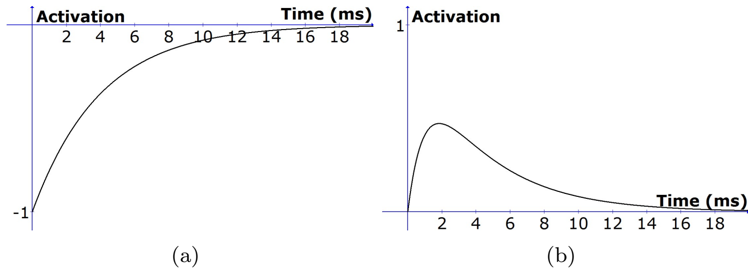


Figure 2.3: Shapes of a possible eta kernel (a) and epsilon kernel (b).

neural networks, the situation is a little different. In traditional neural networks, assigning meaning to the activation of input and output neurons is a quite straightforward process. If for instance, the task at hand is to decide which painter painted a specific painting, one could let the input neurons represent specific features of the painting. This process is called *feature selection*, and see for instance [34] for a painter recognition application of neural networks. The output neurons could then represent individual painters. This straightforward interpretation makes applying a learning procedure useful, as one can steer the network to perform better on the task.

For spiking neural networks, it is a lot harder to interpret the behavior of individual neurons. Where neurons in a traditional network will, for any given input, simply output a single value, spiking neurons receive a continuous input, and produce a continuous output consisting of a number of spikes. In this output, not only the number of spikes is important, but also the timing of individual spikes, and the combination of timings of multiple spikes, possibly of different neurons. Thus, interpreting this neural code is far from trivial [24].

If interpreting the behavior of neurons is very hard, it is even harder to let a spiking neural network learn to perform the desired behavior. Thus, though supervised learning methods for spiking neural networks do exist [35], unsupervised learning methods are usually used. We shall here focus on a particular unsupervised method: Spike-Timing Dependent Plasticity (STDP) [24, 26, 33, 36].

In STDP, given a *presynaptic* neuron j that sends spikes to a *postsynaptic* neuron i, a weight modification is applied for every pair of pre and postsynaptic spike. Both the size and the direction of the weight change are determined by the difference in timing of the pre and postsynaptic spike $t(j) - t(i)$. Experimental evidence has shown that this type of learning procedure is also present in the brain. Though the dependency of size and direction of weight change on timing of pre and postsynaptic spikes as depicted in Figure 2.4 is most common, different dependencies have been found across different types of neurons [26].

To prevent the weights of connections to grow unbounded to either very large positive or negative values, some measure needs to be added to keep the weights within some range. Note that in the function as depicted in Figure 2.4 such a measure is not present. The range in which weights are allowed to fluctuate is usually from 0 to 1. One way to enforce this range is to multiply the weight change with $1 - w_{ij}$ if the connection is strengthened, and w_{ij} if the connection is weakened. This is a so-called soft restraint. A hard restraint would be to simply cut a weight back to either of the boundaries of the allowed range if it

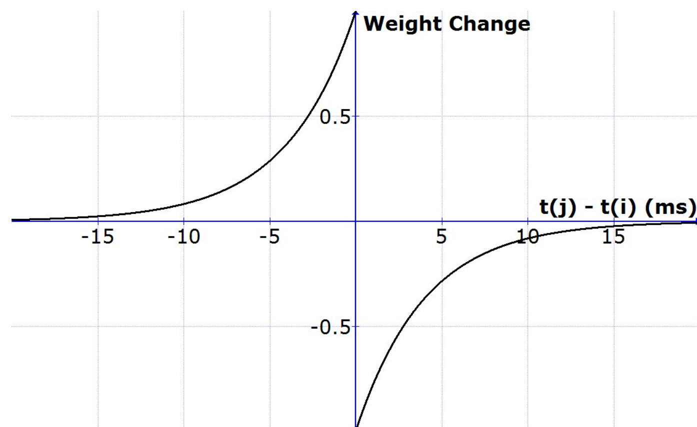


Figure 2.4: A typical form of the STDP learning function. If the presynaptic spike precedes the postsynatpic spike the connection is strengthened, else it is weakened.

grew out of the boundaries.

One can see that this is a Hebbian learning procedure, as described earlier, as indeed connections between neurons that are efficient at making one another spike are strengthened. However, the weakening of neurons if they spike too late introduces a form of competition between neurons. Presynaptic neurons will have to compete over control of the spike timing of postsynaptic neurons [36]. The result is that in a spiking neural network in which this learning procedure is applied, clusters of neurons evolve that spike highly synchronously.

2.4 Features of Neural Networks

We will now discuss several features of neural networks that make them especially fit for our needs. We start by noting that since traditional neural networks have no natural way of dealing with input that is spread over time, we will be borrowing heavily from spiking neural networks. In spiking neural networks, time is an integral factor in the way the neurons work. This is desirable, since we will be using sensor readings over time to perform the tasks.

2.4.1 Distributed

Artificial neural network operate, by their very nature, in a distributed and parallel fashion. The artificial neural network's largest single unit of computation is the neuron, which does little more then calculating its own activation based on the incoming activations from lower layers in the network, and updating the weights of the connections it has to these layers. It does so independent of neurons that it is not connected with. Yet, it has been shown that neural networks are computationally as powerful as Turing machines [22]. However, because all neurons from the same layer perform their computation in parallel, the neural network is potentially much faster than a traditional sequential computer. This feature is of course not present in a neural network implemented by a computer,

since a computer can only simulate the neural network in a sequential manner. By viewing the entire sensor network as a neural network, we thus inherit the distributedness of neural networks.

2.4.2 Learning

A second feature is that neural networks are able to learn how to perform a certain task, as described above. Theoretically a computational system is a mapping of an input space to an output space. In traditional computing this mapping has to be designed by the designer of the system. While in some cases this can be desirable, for instance because you want absolute control over what the system will do, it can become unrealistic when designing this mapping is too hard for a human being to do. While for a task like deciding whether a painting is a van Gogh or not it may be feasible to design this mapping, if the task is to decide which of ten painters painted a painting, it may not be feasible. Neural networks, if trained correctly, can learn this mapping.

Though training a traditional neural network to perform a certain task is quite straightforward, for spiking neural networks this is less so, as we argued above. Still, we will be using a learning procedure that is similar to the STDP learning procedure outlined above, to learn the typical behavior of agents. However, we will take great care in designing the network in such a way that neurons learn in a meaningful manner.

2.4.3 Robustness

Another feature that neural networks have is that they generally are robust to noise in the input and failure of individual neurons. Because every input neuron is connected to many neurons in the next layer, and every neuron in this next layer is connected to many input neuron, it does not really matter if one or a few input neurons have wrong activations. These wrong activations will be compensated and the consequences will be limited. This stands in contrast to traditional sequential computational methods, where one wrong bit can cause system failure [28]. Also, if one removes a neuron from a neural network it will typically only perform marginally worse, since only a tiny bit of the mapping it has learned is removed. This phenomenon is also called *graceful degradation*. Moreover, if one trains the network again, other neurons will simply learn to fill the gap. This in contrast to for instance computers that simply do not work if a system component is broken.

We have identified robustness as a challenge for any application being developed for a wireless sensor network. Our hope is that by viewing the entire sensor network as a neural network we will inherit the property of robustness the same way we inherit the property of distributedness.

2.4.4 Security of Data

One major challenge to the use of ambient intelligence systems is the issue of privacy concerns. Ambient intelligence systems allow for the collection of personal data on an unprecedented scale. This data can of course be used to both benign and malicious end, but between the two there is a large gray area, making it unclear when use of this data becomes misuse. If it hard to decide

when use becomes misuse, it is even harder to protect the data from this [17]. Thus, if we explicitly store data in whatever way, the system is vulnerable to the misuse of data. Neural networks do not store data explicitly, and, moreover, whatever they have learned is completely distributed over the network. Data that enters the network is constantly being used to generate output, that will have to be used on the spot. Patterns that the network learns are stored in the form of many values that are distributed over the network, and that do not mean anything by themselves. Thus using the paradigm of neural networks inherently protects data that is gathered, and patterns that are learned, from theft or misuse.

Chapter 3

The Tracking, Prediction and Identification Algorithm

In this chapter we shall present an algorithm that is capable of tracking agents in an environment, predicting their locations in the future, and identifying the agents based on the paths they travel in the environment. The algorithm is a neural network, and will include a learning mechanism to facilitate prediction and identification as well as to increase the system's robustness to noise. First, we shall introduce the assumptions we make, from which we start working on the algorithm. We will then first discuss related work on the tracking task, present our own approach, and develop the tracking algorithm. We will then do the same for the prediction task and finally the identification task. We will end the chapter by discussing the proposed algorithm.

3.1 Assumptions

Due to the limited scope of this thesis, we can not develop every part of the sensor network. Thus, we are forced to make a number of assumptions.

We assume that a number of nodes are distributed throughout the environment in some random but even way. Every node carries a binary and anonymous sensor, as discussed in section 1.2. Each node has its own system clock, is capable of some computation and able to store a limited amount of data. We shall not analyze the exact requirements the algorithm poses to the hardware of individual nodes. We do, however, conjecture that with sufficient optimization of the algorithm, these requirements will not be an issue in choosing hardware for an implementation of the algorithm. We will explore this issue a bit further in chapter 5.

We assume that each node is capable of broadcasting messages over a short distance (i.e. < 10 meters). We call this its *broadcasting range*, every other node within this broadcasting range a *neighbor*, and the set of neighbors the *neighborhood* of a node. Broadcasting here means sending a single message in every direction that can be received by every node within the broadcasting

range. We assume that nodes are capable of sending such a message at least once a second, and can receive multiple such messages per second. We assume that such a message can contain, apart from a unique id of the sending node, at least some ten bytes of data.

We further assume that during a network initialization phase these nodes can identify any other nodes within their broadcasting range and can determine the physical distance between itself and other nodes. This could be done for instance by using the signal strength of a message received from another node. Another option would be to have every pair of nodes compare their neighborhoods with one another. They could then, with some accuracy, derive the distance between them from the amount of shared neighbors.

We introduce a common timescale consisting of equally sized time steps, for instance one second. This will be the scale on which our algorithm shall operate. Note however, that we do *not* introduce a global clock in the network. We demand that nodes operate in time steps of the same length, not that they always share the same exact time. Had we demanded this, we would have to devise some way to keep all nodes in the entire network synchronized, and this is not something we wish to be bothered with, if it can be helped.

One final assumption concerns the agents in the environment. We assume that agents move through the environment at a constant speed, that is known a priori. We will give directions on possible improvements to the algorithm so that this assumption may be relaxed when discussing the algorithm in chapter 6.

3.2 Tracking

We will now first review related work on tracking in wireless sensor networks before presenting our own approach and developing the algorithm.

3.2.1 Related Work

In [11, 37, 38], probabilistic algorithms are proposed to track moving objects in an environment. In probabilistic algorithms, the goal is to find a mapping of sensor data to a vector of targets with associated location that maximizes a posterior probability[1]. In other words: given a set of input data, find the set of real world events that was most likely to give rise to this set of input data. These algorithms focus on distinguishing sensor readings caused by agents from sensor readings caused by noise, and on associating sensor readings to the correct agent, if multiple agents are in the environment. This makes these algorithms robust to noise.

The distinction between a sensor reading caused by an agent and a sensor reading caused by noise is made based on the readings of multiple sensors. A fundamental observation underlies this: agents or objects move through the environment, rather than that they teleport. Thus, they will typically cause a trail of sensors readings over time, rather than isolated sensor readings. Distinguishing between sensor readings caused by an agent and sensor readings cause

[1] An extensive description of probabilistic models in general is beyond the scope of this thesis, but see for instance [39] for an elaborate introduction.

by noise based on this observation, can thus only be made using multiple sensor readings.

The probabilistic algorithms proposed in [37, 38], requires all input data to be present in one point. Thus, these algorithms are centralized by nature. As we saw already, in wireless sensor networks using a central processor typically is not feasible. In [11] an attempt to solve this was to chop up the sensor network into geographical regions, each with an own "supernode", that is more capable then the regular sensor nodes. Sensor nodes were assigned to their nearest supernode, thus creating a number of clusters. The supernode performs all computation of the cluster, using all sensor data that nodes in the cluster gather. However, in a wireless sensor network that is designed to be cheap and ad-hoc deployable even in hostile environments, the presence of "supernodes" is a somewhat daring assumption.

Another kind of algorithm that is frequently suggested are geographical interpolation algorithms [12, 18, 40]. In these algorithms multiple nodes cooperate to determine the location of an agent by interpolating the positions each node sensed an agent at. Since nodes only cooperate if they are near one another and they sense an agent at the same time, this approach is a lot more distributed in nature.

All the proposed algorithms mentioned above assumed the availability of sensors that are capable of determining the distance to an agent. Also, the sensors are assumed to be able to do so in every direction, within a certain radius. However, specifically what kind of sensor is capable of doing this is not discussed. In [20], a tracking algorithm was proposed that used only binary sensor data: every sensor could output only one bit of data at a time. It was shown that tracking agents was possible with this minimalist approach, but at the expense of some strong assumptions. The single bit that sensors could output was whether an agent was moving to the node or away from the node, meaning that sensors still had to be able to detect the distance to an agent. Also, the data was assumed to be gathered in one central processing unit. This was assumed to be feasible, since nodes only had to send messages containing one bit of information. In a large network however, it is not just the size of messages that challenges the communication channels, but also the amount of messages that have to be routed to the central station. The amount of messages in particular will tend to clog up the network near the central unit [21].

3.2.2 Our Approach

We will use the observation that agents will typically cause a trail of sensor readings over time. However, in contrast to the probabilistic algorithms described above, we will create an algorithm that is distributed by nature. We let every node in the network gather evidence for the claim that an agent is currently present at the location of the node in the environment. To do this, nodes use each other's sensor readings, but otherwise they gather evidence independently.

To achieve this, we let the network implement an artificial neural network. The neural network has two layers: an input layer and an output layer. Every node in the network implements an input neuron and an output neuron, and the location of these neurons in the neural network is tied to the physical location in the environment of the node that implements them. An input and output neuron are connected if the nodes that implement them are within each others

broadcasting range. Note that an input and output neuron implemented by the same node always are connected. Thus, the physical distribution of the nodes in the environment determines the architecture and connectivity of the neural network that is implemented.

Each time step, a node uses its sensor to set the activation of its input neuron to a value of either 0 or 1 for that time step. This can be done in various ways. The node could take only a single sensor reading, and let it determine the activation of the input neuron (note that since the sensors are binary, this case only requires a one to one mapping form sensor reading to input neuron activation). Alternatively, the node could take multiple sensor readings, and use a threshold function. If the activation of the input neuron is set to 1 we say it *spikes*, and the node broadcasts this spike. The message used to broadcast this spike in principle needs to contain only the id of the node, since if the input neuron does not spike, we do not broadcast this.

Also during the time steps, the node receives zero or more messages from other nodes that tell it an input neuron spiked. The node stores these incoming messages as a combination of the originating input neuron and the time that the message was received. Note that by using the time a message is received we allow the clocks of different neurons to be unsynchronized. Using the activation of the input neuron, and possible incoming or stored messages, the node updates the activation of its output neuron.

We interpret the activations of output neurons as the evidence that a node has gathered for the claim that an agent is currently present at the location of the node in the environment. The higher the activation of the neuron, the more evidence the node has gathered for the claim. Thus we track agents by making sure that at all times, every node knows whether an agent is present at its location in the environment or not.

Thus, every node performs the exact same task of gathering evidence, but is responsible for its own location in the environment. In this way, the tracking algorithm is built up in a truly distributed way: there is no single that performs more computation, has more responsibility, or has to collect more data than any other node in the network.

We are now ready to define the activation function of output neurons. Note that for the tracking task, this is the only component of the neural network that still needs to be defined. We have already defined the neural network's architecture and connectivity, and we do not yet introduce a learning procedure.

3.2.3 Basis

As basis for updating the activation of output neurons we take a linear activation function. The activation A of output neuron i at time step t is the sum of activations A of all connected input neurons j received at time t, multiplied by the weight w_{ij} of the connection between i and j at time t. For the tracking we do not need to have weights associated with connections, but we will later on for the prediction task. Thus, we introduce them now, and simply keep them fixed on a value of 1 for now.

$$A_i(t) = \sum_{j=1}^{n} A_j(t) * w_{ij}(t) \tag{3.1}$$

Let us look at the interpretation of this activation function. Suppose the sensor of node A senses an agent at time $t = 1$. It sets the activation of its input neuron to 1, broadcasts a message, and then updates the activation of its output neuron. If no other node senses an agent at time $t = 1$, the activation of the output neuron will be 1. Suppose the message that node A broadcasts is received by a node B, located five meters away from node A, also at $t = 1$. Node B will then also update the activation of its output neuron, and if it receives no other messages, the activation of its output neuron will also be 1. Thus, node A and B gathered the same amount of evidence at $t = 1$, while it was node A that sensed the agent.

The next time step, the activation of the output neuron will be reset, and thus the effect of the sighting of the agent at $t = 1$ is gone. Thus, if at $t = 2$ no sensor sense the agent, the output neurons of node A and B again gather the same evidence. However, since node A did sense the agent at $t = 1$, we can be sure that it still is at least in the vicinity at $t = 2$.

Thus, we can define two problems arising from equation 3.1: (1) nodes forget, from one time step to the next, that an agent was near, and (2) nodes treat every sighting of an agent they receive a message about as full evidence that the agent is at their location.

3.2.4 Memory

We solve the first problem by giving output neurons memory. Every time step, we let output neurons retain a portion of its activation of the previous time step. This means that evidence that was gathered at some time step does not become void instantly, but rather decays in relevance over time. We do this by introducing a *decay-factor* $0 < \gamma < 1$ to the activation function, that reserves a portion of the activation of the previous time step (see 3.2a). We can rewrite this recursive definition to a non-recursive one by summing over all time steps t' starting at $t' = 0$ up to $t' = t$, and multiplying the input from every time step by the appropriate factor $\gamma^{t-t'}$ (see 3.2b). In this alternative definition, it is clear that the effect of every individual input spike on the activation of the output neuron is maximal when it arrives, and then decays. A decaying effect of individual spikes is also used in spiking neural networks, though the shape of the decay function differs. We will adopt this latter definition because it will allow us to solve the second problem mentioned above in a straightforward way.

$$A_i(t) = \sum_{j=1}^{n} A_j(t) * w_{ij}(t) + \gamma * A_i(t-1) \quad (3.2a)$$

$$A_i(t) = \sum_{j=1}^{n} \sum_{t'=0}^{t} A_j(t') * w_{ij}(t') * \gamma^{t-t'} \quad (3.2b)$$

Computationally, equation 3.2a is less expensive, since nodes will only have to preserve the activation of the output neuron into the next time step. Equation 3.2b requires nodes to store all messages it ever receives, which will quickly become infeasible because we assume nodes to be able to store only a limited amount of data. However, because we defined $0 < \gamma < 1$, if $t - t'$ becomes larger for a particular message, $\gamma^{t-t'}$ will become smaller and approximate 0. Thus,

it is not necessary to store messages forever. In fact, if we define a threshold on how small we allow $\gamma^{t-t'}$ to become for a particular message, we have a clear and decisive rule for how many time steps messages have to be preserved, which depends only on γ. However, since how big an issue this is depends on the choice of hardware for implementing the network, we will not define such a threshold here.

3.2.5 Delay

Though equation 3.2b gives the output neurons a form of memory, the second problem we identified is still present. If the input neuron on node A spikes, the output neurons of nodes A and B still treat this spike in the exact same way. This is a problem, because if node A saw an agent, this means the agent is near node A, and not near node B. However, if node B is not very far away from node A, a sighting at node A could indicate that the agent will be at node B some time in the future. To illustrate this, consider the following situation. Node A is located at the beginning of a corridor with a length of five meters, and node B is located at the end of the corridor. If node A sees the agent, and the agent walks in the direction of node B, then the sighting of node A indicated that the agent will be at the location of node B some time in the near future.

Thus, when an input neuron spikes, the impact of the corresponding message on the activation of an output neuron should be maximal when the agent may be expected to arrive at the output neurons. We incorporate this by introducing a *delay factor* δ for each pair of input and output neuron, that reflects how long it would take an agent to travel from the location of the input neuron to the location of the output neuron. This delay factor causes the impact of a spike of the input neuron on the activation of the output neuron to build up over δ time steps. Call v the move speed in meters per second of agents, d_{ij} the physical distance between an output neuron j and input neuron i, and T the size of individual time steps in seconds. Since we assume that the move speed of agents is known beforehand, and that nodes can determine the physical distance they are apart, we can then define the delay factor δ_{ij} between i and j as:

$$\delta_{ij} = \frac{d_{ij}}{v * T} \tag{3.3}$$

The activity updating function then becomes:

$$A_j(t) = \sum_{i=1}^{n} \sum_{t'=0}^{t} A_i(t') * w_{ij}(t') * \gamma^{\left|t-t'-\delta_{ij}\right|} \tag{3.4}$$

If we compare this activation function to the Spike-Response model introduced in chapter 2, we see that it is an activation function that consists of only an ϵ kernel. The ϵ kernel itself is quite distinct from the one introduced in equation 2.7b however. One reason for this is that our kernel neatly incorporates delays between neurons, whereas the kernel in equation 2.7b does not. Moreover, as we will further explore in chapter 5, our kernel is computationally much less expensive.

To track successfully, the algorithm should be aware of the location of an agent as much as possible. This means that (1): the output neuron of the node

that is nearest to the agent should have a high activation, or, in terms of interpretation, enough evidence to conclude that the agent is present at the location of the node. And (2): that output neurons of nodes at locations where no agent is present should not have a high activation, or, in terms of interpretation, not enough evidence to conclude that the agent is present.

Thus, we introduce a threshold θ that determines how high the activation of an output neuron must be for it to conclude that an agent is present. If the activation of an output neuron is equal to or higher than θ during some time step, the neuron *spikes*. Note that this use of a threshold is identical to the way in which neurons in spiking neural networks spike. However, our activation function does not depend in any way on the last time a neuron has spiked, as in spiking neural networks.

The actual value that θ should have depends on a number of factors. The factors include the theoretical maximal activation a neuron is able to achieve, the typical activation an output neuron will achieve when an agent is present. Other factors are specific demands on the performance of the system. We shall explore this issue further in chapter 4, when we will test the performance of the algorithm in a simulation environment. For now we will use an example to illustrate how the algorithm is able to successfully track agents in an environment.

Example 1. *Imagine a mini-environment measuring three by two meters, where each square meter in the environment contains a node. An agent moves about in this grid-world, each time step occupying exactly one square meter, and moving about with a speed of one meter per second. The agent can only walk horizontally and vertically. This is illustrated in Figure 3.1.*

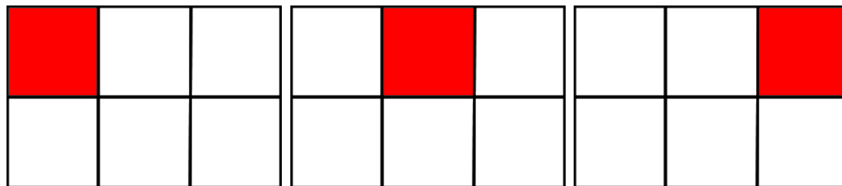


Figure 3.1: An agent moves about in a grid-world environment with a speed of one meter per second.

Now suppose we have set our decay factor γ *to 0.5. The distance between nodes is the Manhattan distance*[2] *between them. Suppose the agent started walking at* $t = 1$. *We will now compare the activations of the output neuron at squares* $(3, 1)$ *and* $(2, 2)$, *to see if indeed the output neuron at* $(3, 1)$ *has a higher activation than the output neuron at* $(2, 2)$ *at* $t = 3$. *From Figure 3.2 we see that the three inputs all have their maximum impact on* $(3, 1)$ *at* $t = 3$, *together delivering an activation of* 3. *On* $(2, 2)$ *however, the maximum effect of the last input is delayed, causing the activation of* $(2, 2)$ *at* $t = 3$ *to be smaller than that of* $(3, 1)$. *Thus, with* $2.25 < \theta \leq 3$, *we would have the desired effect of output neuron* $(3, 1)$ *spiking, and neuron* $(2, 2)$ *not spiking.*

[2] The Manhattan distance measure is inspired by a grid layout of the environment, where one can only walk horizontally or vertically. This typically is the case in many American cities, where streets are aligned in a grid-like way, and the squares between streets are buildings. The Manhattan distance between two vectors is calculated by summing over the absolute distances between all components of the vectors.

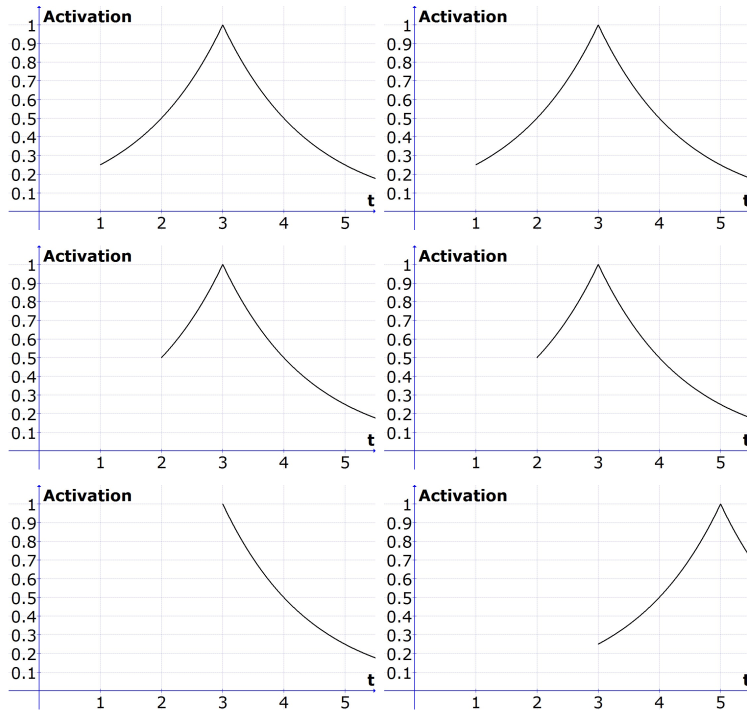


Figure 3.2: Left: the effect that the input neurons at location (1,1), (2,1) and (3,1) have on the output neuron at location (3,1). Right: the effect these input neurons have on output neuron (2,2).

3.3 Prediction

In this section we will present a learning procedure that will allow the sensor network to predict the movements of agents in the future.

3.3.1 Related Work

We have introduced prediction as a task that is necessary for the system to be able to perform higher level tasks such as controlling the environment and geofencing. In the literature, prediction of the movements of agents is often introduced as a means to reduce power consumption of nodes. If nodes can predict the course of an agent, they can alert nodes along the trajectory of the inbound agent. Thus, nodes need only sense their environment if an agent is inbound, saving energy.

One frequently used strategy for predicting the movements of agents is to apply linear or polynomial curve fitting on recent geographical positions of an agent. This results in a trajectory that the agent is predicted to travel in the near future. One assumption that has to be made for this strategy to work is that nodes know at least the physical location of other nodes, relative to their own physical location. Another assumption is that agents always walk in a straight line or in a higher order curve. Though this last assumption may be true in open environments like a forest or a desert, it is typically not true in

buildings or cities, where the environment forces agents to make a lot of turns.

In [19] a different approach was taken, that would solve this problem. The observation was made that agents typically have a fixed movement pattern or motion model. The motion model of an agent is a model specifying the behavior of the agent in the environment. It could for instance include information on sequences of locations the agent will visit, information on how long the agent will take to travel from one location to another, or at what specific time an agent will travel a specific route.

The motion model of a bus for instance, could specify what route it will travel in a city, and at what time it will arrive at specific locations on the route. The motion model of people in a factory could specify that people walk between their work space and the canteen at fixed times during the day.

Thus, a method was introduced to learn the particular motion models of agents, and to spread these patterns through the network so that nodes could predict the movements of agents. It was assumed that agents would themselves gather detailed information on the patterns they travel, for some period of time, and deliver this information to a series of supernodes, located at the outer ring of the network. These supernodes would apply a data-mining technique to extract the patterns from the data, and disseminate these patterns to the rest of the sensor network. The assumption that agents will gather information themselves may be feasible when the agents are buses that can easily be outfitted with the required technology. But in an environment where the agents to be tracked are animals, or enemy military forces, it is not.

3.3.2 Our Approach

To enable the algorithm to predict the location of agents in the future, we will introduce a learning procedure that will allow the algorithm to learn limited motion models of agents. The motion models we learn are limited because they only specify whether agents travel from one location to another location. We do not need to take into account the time they will take to travel from one location to another, because we have assumed we know the move speed of agents and the distance between nodes.

To see how the learning of motion models will allow for prediction, we first have to define what counts as a prediction in our algorithm. Recall that if a sensor senses an agent, the corresponding input neuron has its activation set accordingly, and that the node sends a message to other nodes. The output neurons of other nodes then use this message to update their own activity. We had already observed that the receiving output neurons should not interpret this message as meaning that an agent is present at its location right now, but rather that an agent may be present some time in the future. Thus, we may view the message that is sent by a node to other nodes when its sensor senses an agent as a number of predictions equal to the number of receiving nodes.

It may be obvious that a great deal of these predictions are wrong. One obvious problem that causes this is that the messages that nodes send go in every direction, while an agent does not. However, since we assume that nodes broadcast their messages, we can not cull these wrong predictions by limiting the amount of recipients of a message. We can, however, let output neurons learn to ignore messages from sensors that consistently provide wrong predictions, by reducing the weights of connections with the corresponding input neurons.

By initializing the weights of connections to 1, we have implicitly assumed that agents can and will move from one node to any other node that is near. If we let every individual node learn, for each neighboring node, whether agents travel from the location of the neighboring node to its own location, we obtain a distributed motion model. For every connection between an input neuron and an output neuron, a weight of 1 means agents do travel from the location of the input neuron to the location of the output neuron. A weight of 0 means agents do not travel from the location of the input neuron to the location of the output neuron.

3.3.3 Learning Procedure

In chapter 2, we have seen the learning procedure called STDP. We will implement a similar learning procedure. We would like to strengthen connections over which correct predictions are sent, and weaken connections over which incorrect predictions are sent. We say a prediction is correct if before the effect of the corresponding message on the activation of the output neuron has worn out, the output neuron has spiked. We say a prediction is incorrect if the output neuron did not spike before that time.

We can now introduce a limit τ on how small the effect of a message on the activation of the output neuron may become, such that we still call the prediction successful. In combination with the decay-factor γ, this τ leads deterministically to a time window in which predictions will be called successful. If a sensor senses an agent, and thus the activation A of the corresponding input neuron j is set to a value of 1 at time t, strengthen the connection between j and an output neuron i if i spikes before $t + \delta_{ij} + \frac{\log \tau}{\log \gamma}$.

$$\text{If } A_j(t) \geq 1 \begin{cases} \text{strengthen } w_{ij} & \text{if } A_i(t') > \theta \text{ for some } t \leq t' \leq (t + \delta_{ij} + \frac{\log \tau}{\log \gamma}) \\ \text{weaken } w_{ij} & \text{otherwise} \end{cases} \tag{3.5}$$

One major difference with the STDP learning method introduced earlier is that in STDP weight modifications only occur if both the pre and the postsynaptic neuron spike (i.e. both the input and the output neuron). In our learning procedure, spiking by the presynaptic neuron, or input neuron, is enough for a weight modification to occur. This is called presynaptic gating [24]. A second major difference with STDP is that in our learning procedure the relative timing of pre and postsynatpic spikes does not determine the size of the modification, whereas in STDP it does.

Instead, we define the strengthening and weakening functions such that the weight itself will determine the size of a modification. Reason for this is that it will allow us to define such a strengthening and weakening function that the only stable values that weights can converge to are 1 and 0. Thus, a weight that converges to a value of 1 indicates that agents do walk the path from the location of the input neuron to the location of the output neuron, while a weight of 0 indicates that they do not. This allows straightforward prediction, based on the learned motion models, while weights that settle on a value of for instance 0.5 would require additional interpretation.

Thus, we introduce the following strengthening (equation 3.6a) and weakening (equation 3.6b) functions, with $0 < \sigma \leq 1$ and $0 < \omega \leq 1$ parameters that

determine the force of the functions. Note that we omit the ij suffix of weights, since these functions do not depend on it.

$$w(t+1) = w(t) + \sigma * \Delta w(t) \tag{3.6a}$$
$$w(t+1) = w(t) - \omega * \Delta w(t) \tag{3.6b}$$

Where:

$$\Delta w(t) = w(t) * (1 - w(t)) \tag{3.7}$$

In Figure 3.3(a) the Δ function of equation 3.7 has been drawn. As one can see, the form of this function ensures that weights with a value of around 0.5 will cause a high delta value, making it impossible for them to stabilize. Rather, under the influence of many incorrect predictions they will quickly converge to a value of 0, and under the influence of many correct predictions they will quickly converge to a value of 1.

In Figure 3.3(b) the strengthening (solid line) and weakening (dashed line) have been drawn for $\sigma = 1$ and $\omega = 1$ in recursive form. One can see that only if the amount of correct and incorrect predictions this weight is influenced by is equal, the weight will not converge. This is because σ and ω have been set to the same value, and thus, these values can be used to bias the weight to allow more or less incorrect predictions before converging to 0.

One last thing remains to be said. For weights that have a value between 0 and 1, our learning procedure implements 0 and 1 as soft restraints on the possible values a weight can attain, as introduced in chapter 2. We have thus far said we would initialize our weights to a value of 1, but with this learning procedure, a weight of 1 not be able to change. Thus, to actually make the learning procedure work, we will have to initialize the weights to a value a tiny bit smaller than 1, for instance 0.99.

Again we will illustrate the working of the learning mechanism with an example.

Example 2. *We assume the same scenario as in example 1: an agent walks in a grid like environment from* $(1,1)$ *to* $(3,1)$*, and exits the environment there. Thus, while the output neuron on* $(3,1)$ *will spike at* $t = 3$*, the output neuron on* $(2,2)$ *will not, despite receiving some input, as we have seen in example 1. We will now assess the consequence this has on the weights of the connections between the input neuron on* $(1,1)$ *and the output neurons on* $(3,1)$ *and* $(2,2)$*. We assume that* $\sigma = \omega = 1$*.*

At $t = 3$ *the output neuron on* $(3,1)$ *spikes, and since the input neuron at* $(1,1)$ *had a part in that, the connection between the two is strengthened:*

$$\begin{array}{lclcl} w_{(3,1)(1,1)}(4) & = & w_{(3,1)(1,1)}(3) & + & w_{(3,1)(1,1)}(3) * (1 - w_{(3,1)(1,1)}(3)) \\ 0.9999 & = & 0.99 & + & 0.99 * (1 - 0.99) \end{array}$$

Simultaneously, as the output neuron on $(2,2)$ *does not spike, the connection between it and the input neuron on* $(1,1)$ *is weakened after the effect of the input form* $(1,1)$ *has decayed beyond* τ*:*

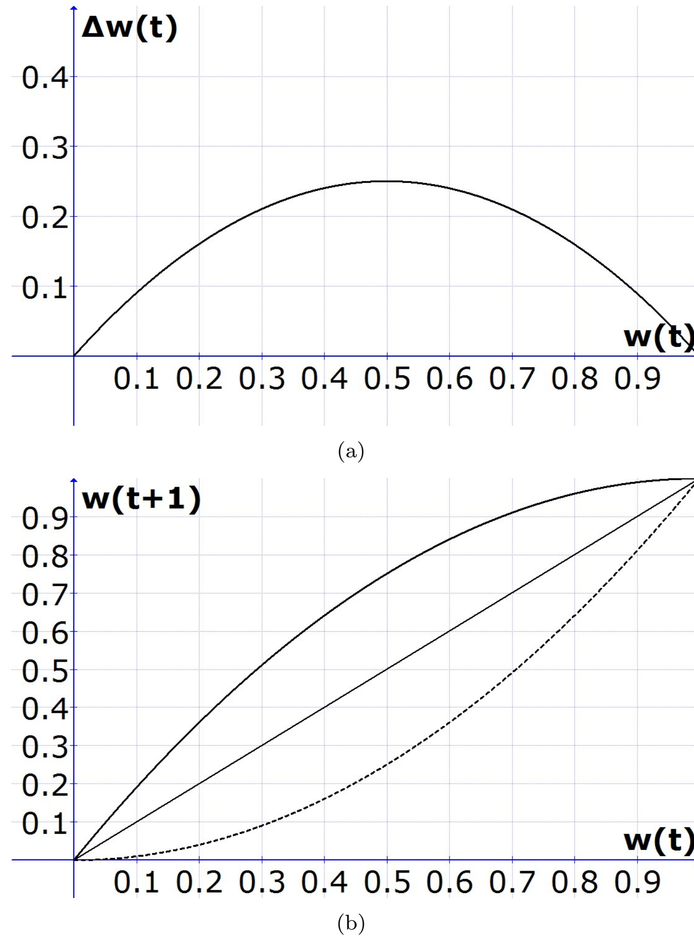


Figure 3.3: Delta function (a) and the strengthening and weakening functions for $\sigma = \omega = 1$ (b).

$$\begin{array}{lclcl} w_{(2,2)(1,1)}(6) & = & w_{(2,2)(1,1)}(5) & - & w_{(2,2)(1,1)}(5) * (1 - w_{(2,2)(1,1)}(5)) \\ 0.9801 & = & 0.99 & - & 0.99 * (1 - 0.99) \end{array}$$

If the agent consequently only walks this path in the environment, in the limit the weight between $(3,1)$ *will become* 1*, and the weight between* $(2,2)$ *and* $(1,1)$ *will become* 0*. Thus, the output neuron at* $(2,2)$ *will learn to ignore the input neuron at* $(1,1)$*, causing the network to generate fewer incorrect predictions.*

3.4 Identification

We will now present an extension of the algorithm so that it is able to recognize a number of classes of agents (for instance: patient, nurse, guest). The current algorithm can then be seen as the special case where there exists only one class of agent. We assume for now that the amount of different classes that may be

present in the environment is known beforehand. Later, in chapter 6, we will discuss this assumption and give ideas on how to relax it.

3.4.1 Related Work

Identification without using identifying senors is in the literature often presented as a task of classifying agents into one of several classes of agents. Two main observations are used to do this. The first observation is that different classes of agents tend to cause different types of sensor readings [41]. A wheeled vehicle, such as a car, will for instance cause a significantly different reading on a seismic sensor then a tracked vehicle, such as a tank. If one is to use the different types of sensor readings to classify agents, one has to verify that different classes of agents indeed do cause different types of sensors readings. This may be a valid assumption for sensors such as seismic sensors, cameras and microphones. However, when employing binary sensors it is simply impossible for sensors to make classify agents based on the type of sensor reading they cause.

The second observation is an extension of the observation we mentioned earlier: that agents will typically have a motion model according to which they move through the environment. Different classes of agents will typically have different motion models, meaning that the patterns in which they move through the environment are different [37]. Thus, if one can identify the specific movement dynamics that an agent is exhibiting, one can use this to determine the class of the agent. The inverse is also true: if one knows the class of an agent, one can use its motion model to predict where the agent is going.

In [37], the identification task was solved by assuming that the motion models of all classes of agents that may be in the environment is known beforehand. It may in some settings be possible to know all these motion models beforehand: if one wants to track buses in a city, then indeed their particular routes are known and available. However, programming all these motion models into the system beforehand deals a heavy blow to the ad-hoc deployability of the system. Assuming that nodes can not store the entire model for every class of agent, one has to give every node a particular part of every motion model that corresponds to the node's geographical position.

3.4.2 Our Approach

Before our algorithm will be able to deal with identity as a property of agent, we will have to introduce the notion of different identities into the algorithm. We do this by giving both input and output neurons a separate activation for every known identity. Also, every connection between input and output neuron will have a separate weight for every identity. Alternatively, we could have introduced for every node a separate input neuron and output neuron for every identity, and connect only pairs of input and output neuron that have the same identity. The two are equivalent, but we shall adopt the former method to preserve the idea that every node implements one input and one output neuron.

As we already use a motion model to predict the movements of agents, we will also use motion models to identify agents. Recall that the algorithm learns distributed motion models by settings weights to 0 or 1. As we now have separate activations and weights for every known identity, we can now learn a motion model for every identity. This way, if an agent walks according to the

motion model of some identity, the activations of the output neurons for that identity will become high enough to spike. We will elaborate on how this works below.

First we need to decide how to get the motion models of different classes of agents into the algorithm. As we have discussed above, though it is possible to assume the motion models are known and to program them into the network, this is highly undesirable. We would rather employ our learning procedure to let the algorithm learn the motion models. However, for the algorithm to be able to learn the motion model for a specific identity, it need to know first which agents are of that identity. Thus we are faced with a problem: we would like the algorithm to learn the motion models of agents so that it is able to identify them, but to do that, it needs to know the identity of agents.

We solve this problem by making a minimal concession to the design decision of using only binary and anonymous sensors. We will assume that in the environment at least one sensor is temporarily available that can in some way determine the class of an agent. When an agent passes an identifying sensors and is identified, we will associate the path the agent consequently walks to its identity. Thus, we essentially introduce a supervised learning phase for the identification task. After some time all motion models are learned and the identifying sensors are no longer needed to identify agents.

Adding even one identifying sensor goes against the argumentation against the use of identifying sensors we gave in chapter 1. However, we are faced here with the fundamental problem that we need some information on the basis of which we can identify agents. Though people may feel adding a single or maybe a few identifying sensors is a violation of their privacy, it is probably less intrusive than having to obtain the motion model of people manually. Thus, it appears that there is no free lunch when it comes to identifying agents, and one will have to make the decision if being able to identify agents is worth temporarily adding an identifying sensor to the network.

3.4.3 Introducing Identity

Giving every input neuron an activation for every known identity allows us to neatly incorporate the newly introduced identifying sensors in the algorithm. We assume that the identifying sensors are more reliable in their measurements than the anonymous sensors, and thus, if an identifying sensor senses an agent, the activation of only the sensed identity is set to θ. This means we assume that if an identifying sensor senses an agent, this alone is enough evidence for output neurons to conclude that an agent is present. If an anonymous sensor senses an agent, the activation of all identities are set to 1, signifying that the sensor "doesn't know" the identity of the agent. If the sensor of a node now senses an agent, a vector containing the activations for all identities of the input neuron is broadcasted.

We update the activation function of output neurons of equation 3.4 to incorporate the newly introduced identities. To calculate the activation A of output neuron i at time t for identity id, we now sum only over the activations for that same identity and multiplied by the weights for that same identity.

$$A_i^{id}(t) = \sum_{j=1}^{n} \sum_{t'=0}^{t} A_j^{id}(t') * w_{ij}^{id}(t') * \gamma^{|t-t'-\delta_{ij}|} \tag{3.8}$$

We now say an output neuron i spikes at time t if for some identity id, $A_i^{id}(t) \geq \theta$. Thus, if the activation of an output neuron for a specific identity is higher than the threshold, the output neuron has gathered enough evidence to conclude that an agent of that identity is present. By allowing output neurons to spike for any number of identities at the same time, neurons can identify agents, if they spike for only one identity. They can also detect an agent, but be in doubt about the identity, if it spikes for some identities. If an output neuron detects an agent but has no information at all on the identity of the agent, it will simply spike for all identities.

Because we have only assumed the presence of some identifying sensors, the majority of sensors will still be anonymous. With equation 3.8, actually identifying an agent will only be possible as long as it is sensed by an identifying sensor. This is a problem because the identity of an agent doesn't change as it moves through the environment. If some output neuron gathered enough evidence for the claim that an agent of identity id is present, we could say that output neuron "knows" the identity of the agent. We would like subsequent output neurons that track the same agent in the future to also know the identity of the agent, even though it may not receive any input from an identifying sensor.

3.4.4 Propagating Identity

We accomplish this by letting output neurons share the information they have concerning the identity of an agent. We add lateral connections between two output neurons if they are implemented by nodes that are within each other's broadcasting range. When an output neuron i' spikes for one or more identities at time t, we denote the set of identities it spikes for by $id_{i'}^{+}(t)$, and the set of identities it does not spike for by $id_{i'}^{-}(t)$. We let the spiking output neuron inhibit the activation of identities in the set $id_{i'}^{-}(t)$ of connected output neurons i. This way, for connected output neurons, the activations of the identities in the set $id_{i'}^{+}(t)$ will be higher than the activations of the identities in the set $id_{i'}^{-}(t)$.

The lateral connections have a fixed weight of 1, as we do not in this thesis define any learning procedure for these weights. Because we keep the weights of lateral connections fixed, we do not need to introduce a separate weight for every known identity. Also associated with every lateral connection between an output neuron i and i' is a delay factor $\delta_{ii'}$, that is set according to equation 3.3. The inhibiting effects take on the form of a negative modification of the activations of identities. This negative modification has a value of -1 multiplied by the weight between the output neurons, and modified by the delay and decay effects, as in equation 3.8.

Thus, to calculate the activation A of an output neuron i for identity id for time t, we add to equation 3.8 the sum of inhibiting effects for that identity. For all connected output neurons i' and all past time steps t', if $id \in id_{i'}^{-}(t')$, we add -1 multiplied by the weight and modified by the decay and delay effects to $A_i^{id}(t)$.

$$A_i^{id}(t) += \sum_{i'=1}^{m} \sum_{t'=0}^{t} \left\{ -1 * w_{ii'} * \gamma^{\left|t-t'-\delta_{ii'}\right|} | id \in id_{i'}^{-}(t') \right\} \tag{3.9}$$

3.4.5 Learning Identity Specific Motion Models

To be able to learn the motion models of different classes of agents, we need to adapt the learning procedure. We now strengthen a weight w_{ij}^{id} between an input neuron j and an output neuron i for a specific identity id if the input neuron spiked for id, and the output neuron also spiked for id within the defined time interval. We weaken a connection if the output neuron did not spike for the same identity. Thus we yield:

$$\text{If } A_j^{id}(t) \geq 1 \begin{cases} \text{strengthen } w_{ij}^{id} & \text{if } A_i^{id}(t') > \theta \text{ for some } t \leq t' \leq (t + \delta_{ij} + \frac{\log \tau}{\log \gamma}) \\ \text{weaken } w_{ij}^{id} & \text{otherwise} \end{cases} \tag{3.10}$$

We shall again use an example to illustrate how the algorithm is capable of propagating the identity of agents through the network, and how by doing this the motion model of the class of agents can be learned.

Example 3. *We assume the same grid-world environment as in example 1. The agent now walks from square* $(1,1)$ *to square* $(3,2)$*, and we change the sensor in square* $(1,1)$ *to an identifying sensor. We assume that there are two known classes of agents,* $id1$ *and* $id2$*, and that the agent is of class* $id1$*. The nodes on the squares* $(2,1)$ *and* $(3,1)$ *are within the broadcasting range of the node on* $(1,1)$*, and thus they benefit directly from the identifying sensor. However, since the nodes on square* $(1,1)$ *and* $(3,2)$ *are not within each other's broadcasting range, their neurons are not connected. Thus, the challenge is for node* $(3,2)$ *to still be able to identify the agent.*

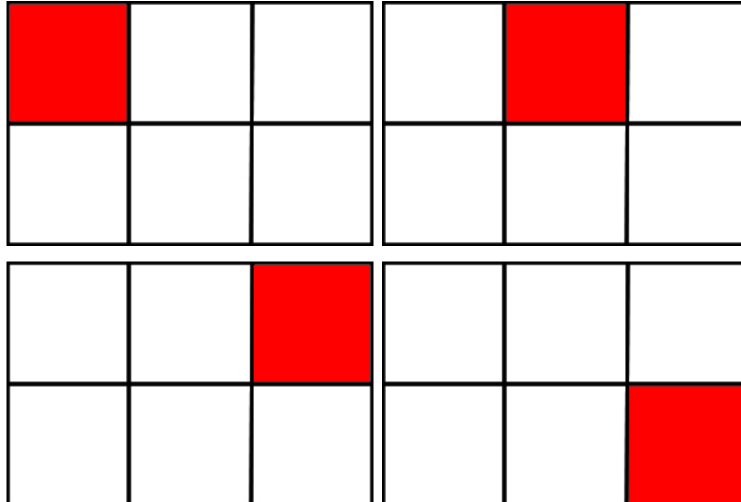


Figure 3.4: An agent moves from square $(1,1)$ to square $(3,2)$ in four time steps.

The output neuron on $(3,2)$ *receives input from the input neurons on* $(2,1)$*,* $(3,1)$ *and* $(3,2)$*. The delay factors make sure that each of these inputs have their maximum effect the moment the agent is present at* $(3,2)$*, at* $t = 4$*:*

$$A^{id1}_{(3,2)} = A^{id2}_{(3,2)} = \begin{array}{lll} 1 * 1 * 0.5^{4-2-2} & \textit{(from } (2,1)\textit{)} & + \\ 1 * 1 * 0.5^{4-3-1} & \textit{(from } (3,1)\textit{)} & + \\ 1 * 1 * 0.5^{4-4-0} & \textit{(from } (3,2)\textit{)} & = 3 \end{array}$$

Thus, based on only the input from input neurons, the output neuron on $(3,2)$ *is not capable of identifying the agent. However, since the output neurons at* $(2,1)$ *and* $(3,1)$ *spiked for* $id1$*, the activation of output neuron* $(3,2)$ *for* $id2$ *gets modified:*

$$A^{id2}_{(3,2)} = 3 + = \begin{array}{lll} 1 * -1 * 0.5^{4-2-2} & \textit{(from } (2,1)\textit{)} & + \\ 1 * -1 * 0.5^{4-3-1} & \textit{(from } (3,1)\textit{)} & = 1 \end{array}$$

Thus the activation of output neuron $(3,2)$ *for* $id2$ *is hampered, output neuron* $(3,2)$ *only spikes for* $id1$*, and is able to identify the agent. Because the output neuron* $(3,2)$ *spikes for* $id1$*,* $w^{id1}_{(3,2)(3,1)}$ *and* $w^{id1}_{(3,2)(2,1)}$ *are strengthened, and because the output neuron at* $(3,2)$ *does not spike for* $id2$*,* $w^{id2}_{(3,2)(3,1)}$ *and* $w^{id2}_{(3,2)(2,1)}$ *are weakened. If this happens often enough, eventually* $w^{id1}_{(3,2)(3,1)}$ *and* $w^{id1}_{(3,2)(2,1)}$ *will become* 1*, and* $id2$*,* $w^{id2}_{(3,2)(3,1)}$ *and* $w^{id2}_{(3,2)(2,1)}$ *will become* 0*. If we then remove the identifying sensor, the node at* $(3,2)$ *is still able to identify the agent, because the output neuron will not gather any activation for identity* $id2$ *at all.*

3.5 Discussion

We have now introduced an algorithm that will operate on a wireless sensor network and that will perform the tasks tracking, prediction, and identification. The algorithm treats the wireless sensor network as one artificial neural network, and borrows heavily from the field of spiking neural networks. Because of this, the algorithm is distributed by design. We employ a learning procedure to learn the motion models of agents, so that nodes do not need to be programmed with data specific to individual nodes.

Nodes perform the tracking task by using the sensor readings from nodes in their neighborhood to gather evidence for the claim that an agent is present at their location in the environment. By implementing a spiking output neuron that is connected to input neurons on nodes in the neighborhood, this is done in a straightforward way.

To let nodes correctly predict the behavior of agents, we let nodes learn the typical behavior of agents. To do this, we employ a learning procedure that takes into account the timing of individual spikes, like the STDP learning procedure that is often used in spiking neural networks.

To let nodes also identify agents, we introduced classes of identities to the algorithm. We let nodes learn the typical behavior for individual classes of agents. Moreover, we let nodes propagate the identity of detected agents through the network, by adding lateral connections between output neurons.

Chapter 4

Validation using Simulation

Now that we have proposed an algorithm for tracking, identification and prediction, it is time to test its performance. Using a simulation environment, we will verify that the algorithm is indeed capable of performing the mentioned tasks. Moreover, we will verify the claim that our neural network algorithm is robust to noise. In chapter 1, we have argued that robustness is a challenge for applications for sensor networks, and used the robustness of neural networks as an argument for viewing the sensor network as a physical neural network. Thus we need to test if indeed our particular neural network algorithm is robust to noise.

To assess the performance of the algorithm on the three tasks and its robustness to noise, we will compare it to a baseline performance. This naive algorithm simply directly uses the measurements of the sensors as output, thus rendering it incapable of predicting the movements of agents, and making it highly susceptible to noise.

We shall first introduce the simulation environment, then present the experimental setup, we will subsequently give the results and discuss them.

4.1 Simulation Environment

The simulation consists of an eight by eight grid, representing an environment. The grid is divided into four rooms and a corridor to make it resemble a building; the environments in the scenarios that drive the development of this algorithm will typically be buildings. See Figure 4.1 for an image of the grid.

On each of the 64 squares of the grid we place one network node. The sensors of the nodes can sense the presence of an agent, but only on their own square. For the testing of the performance of the algorithm on the identification task, we change the sensor of the node in the top left corner of the grid into an identifying sensor. The identifying sensor can sense the identity of an agent, but again only on its own square. The network nodes have a broadcasting range of two squares in Manhattan distance as in example 1. This broadcasting distance is not hampered by walls.

We introduce three agents, each with a different motion model. The “Random Goal” agent randomly chooses a square in the environment and sets it as its goal. It will then take a shortest route to its goal and, once it’s there, select

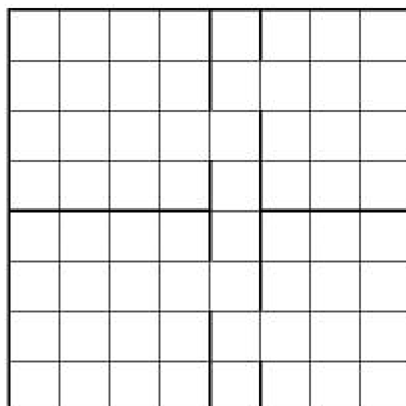


Figure 4.1: The simulation environment: an eight by eight grid containing four rooms and a corridor connecting the rooms.

a new goal, and so on. There are two "Wall Hugger" agents, which always stick to the wall of the environment, and walk around in circles. One Wall Hugger agent walks clockwise, the other walks counter-clockwise.

Since the motion model of the Random Goal agent contains a random element, we expect that predicting the movements of this agent will be a hard task for the algorithm. For this agent, the only fixed component in its motion model is that it has to adhere to the layout of the environment. Thus, the performance of the algorithm on learning to predict the movements of this agent, will show us if the algorithm is capable of learning the layout of the environment from the movements of an agent. We will use the two types of Wall Hugger agents to test if the algorithm will indeed learn to identify agents based on their motion models alone.

We let the simulation run in discrete time steps. Every time step, an agent moves one square, either horizontally or vertically. Thus, agents are not allowed to move diagonally, and do not stand still. Also every time step, the sensor of every node takes a reading and the node broadcasts the activations of its input neuron. Then the output neurons of every node calculate their activations, and if an output neuron spikes, the node then broadcasts this for identification purposes as introduced in chapter 3. Finally nodes may apply the learning procedure to update the weights between input and output neurons.

Last, nodes may cause two types of noise in the environment. We introduce two parameters, P-noise and R-noise, that govern this noise. The first, P-noise, is the probability of a false positive, i.e. the chance sensors detect an agent during a time step, even though no agent is present. This parameter is called P-noise because it it highly related to the performance measure Precision, which we shall introduce below. The second parameter, R-noise, is the probability of a true negative, i.e. the chance sensors do *not* detect an agent during a time-step, even though an agent is present. This parameter is related to the Recall performance measure we shall also introduce below. Setting both P-noise and R-noise to 0 creates a noise-free environment, setting them both to 1 turns the sensors from detectors of the presence of an agent to detectors of the absence of agents.

4.2 Experimental Setup

In this section we will describe the experimental setup: the performance measures used to measure the performance of the algorithm, the parameter settings of the algorithm we used, and the specific experiments conducted.

4.2.1 Performance Measures

To measure the performance of the algorithm on the tracking task, we introduce three performance measures: *Precision*, *Recall* and *PDistance*. Both Precision and Recall are measure well known in the field of information retrieval [42], and we have tailored them to our needs. In information retrieval, *Precision* measures what percentage of documents retrieved on a specific query indeed proved to be useful. We define Precision to measure what percentages of spikes of output neurons of the algorithm was correct. An output spike is correct if at the time of the spike an agent was present at the location of the spiking output neuron. Note that we do not take identity of agents into account here.

In information retrieval, *Recall* measures, for a specific query, what percentage of the existing documents relevant to the query were retrieved. We define Recall to measure the percentage of time steps at which the algorithm was able to determine the location of the agent. We say the algorithm was able to determine the location of an agent if at a particular time step, the output neuron of the node at the location of the agent spiked.

Notice that there is a subtle interplay between Precision and Recall. Output neurons could spike all the time to achieve a high Recall, but this would result in a dramatically low Precision. Similarly, output neurons could spike only if it is totally certain that the agent is present, achieving a high Precision, but this would likely result in a low Recall.

PDistance is a new measure. It measures the average Manhattan distance between the location of an agent at a specific time and the location of erroneous spikes at that same time. For instance, if an agent is at square $(2, 2)$ at time t, and the output neuron at square $(1, 1)$ spikes, this is an erroneous spike, and it has a Manhattan distance of 2 to the actual location of the agent. The intuition behind this measure is that how bad an erroneous spike is, is not a black and white case, but rather depends on the distance of the erroneous spike from the actual location of an agent. An erroneous spike in the near vicinity of an agent might, in an actual application, at least for instance guide a nurse to the correct room where an emergency takes place. In such a case the erroneous spike is less of a problem than an erroneous spike that sends a nurse to the opposite side of the building.

We introduce a fourth and final performance measure to assess the prediction capabilities of the algorithm, called *PredPrec*. PredPrec measures what percentage of predictions the algorithm makes prove to be correct. Recall that in chapter 3 we defined a prediction to be the message a node sends to another node if an activation of its input neuron is 1 or higher. We add to this definition the constraint that the weight of the connection between the input neuron and output neuron must be higher then 0.1. If this is not the case, we can effectively say the output neuron has learned to ignore the input neuron, and thus we do not count it as a prediction anymore. For defining when a prediction is correct, we adopt the same definition as in the learning procedure introduced in chapter

3. Thus, a prediction is correct if the receiving output neuron spikes before the effect of the message has decreased to a value of τ.

4.2.2 Parameter Settings

We set the time steps in which the algorithm runs to be the same as the time steps the simulation runs in. Thus, we say nothing about the actual duration of individual time steps, and this is not required.

In the algorithm, we set $\gamma = 0.5$, $\sigma = 1.0$, $\omega = 0.25$ and $\tau = 0.125$. The combination of γ and τ determines that a prediction at time t only is correct if the receiving output neuron spikes before:

$$t + \delta_{ij} + \frac{\log \tau}{\log \gamma} = t + \delta_{ij} + \frac{\log 0.125}{\log 0.5} = t + \delta_{ij} + 3 \tag{4.1}$$

This window for correct predictions is relevant for the learning procedure, as well as for the PredPrec performance measure. The combination of σ and ω determines that the strengthening function is four times as powerful as the weakening function. This means that every time a connection is strengthened, this can be undone by roughly four applications of the weakening function. Thus we are quite lenient to incorrect predictions, and only punish connections over which consistently wrong predictions are being sent.

To set the δ_{ij} for every pair of input neuron j and output neuron i, we define the physical distance between the two as the Manhattan distance in the grid between the nodes that implement the neurons. We then use equation 3.3 to set every δ. Because the move speed of agents is one square per time step, and the time steps in which the algorithm operates are equal to the time steps of the simulation, this entails that the δ_{ij} for every pair of input and output neuron is simply equal to the Manhattan distance between the two.

Threshold θ

The threshold for output neurons to spike deserves some extra attention. We will initially set θ to a value of 2.75 for every neuron. With the broadcasting range of nodes we have defined, in principle an output neuron will receive a message from three different input neurons when it should spike for the presence of an agent (like the neuron on $(3, 1)$ in example 1). Thus θ should have a value of at most 3, and we set it a little lower to allow for minor deviations from this typical scenario.

However, if we introduce noise into the environment, this typical scenario will not be that typical anymore. Rather, if we add P-noise, output neurons may receive messages from more than just three input neurons. Also, neurons that should not spike, because the agent is not present at the location, may still be able to acquire enough activation to spike. Thus, with the addition of P-noise, one would like to have a higher θ.

If we introduce R-noise, it may happen that a neuron that should spike, is not capable of gathering enough activation to do so, because some input neurons fail to deliver input. Lowering θ to allow the output neurons to spike in these situations will at least raise the Recall score of the network, but will dramatically affect the Precision. With a lower threshold it becomes easier for neurons that should not spike to still gather enough activation to spike.

Thus, while testing the performance of the algorithm in a noisy environment we need to adopt a variable θ. We set θ to a low base value, that allows neurons to spike in case R-noise causes input neurons to fail to deliver input. If output neurons have an activation higher than θ, they exchange their activations, determining which output neuron has the highest activation, so that only that neuron would spike. While testing the performance of the algorithm in a noise-free environment we use a fixed $\theta = 2.75$.

4.2.3 Experiments

We will test the algorithm in runs of 1000 time steps. Each run, the environment will be either noise-free (P-noise = R-noise = 0), contain P-noise (P-noise = 0.01, R-noise = 0) or contain R-noise (R-noise = 0.05, P-noise = 0). Each run, only one agent will be present in the environment at any time, and the type of the agent will not change during the run, except when we test the identification performance of the algorithm. We will then alternate between the clockwise and counter-clockwise Wall Hugger agents, and let them take turns in walking one circle. Thus, only in the identification experiments do we instantiate the algorithm for two known identities.

In all settings, we will test the algorithm both in untrained and trained form. The trained algorithm for a specific experiment has been trained for 5000 time steps in a noise-free environment on the same agent as it will deal with in the actual experiment. For the identification experiments the algorithm is trained on the alternating Wall Hugger agents. During the actual experiments the algorithm is “frozen”, meaning no learning procedure is applied.

In the experiments for tracking and prediction, we measure the Precision, Recall, PDistance and PredPrec scores of the algorithm. We will compare the scores of the algorithm with baseline performance scores on each tasks. For tracking, the baseline performance scores are generated by a naive algorithm where the output is identical to what the sensors measure. For prediction, the baseline performance is generated by the untrained algorithm, since it is not capable of predicting accurately yet.

For identification we simply assess whether the algorithm is capable of correctly determining the identity of the agent currently in the environment from its motion model. Since there is no naive way of learning a motion model to determine the identity of an agent, we will not compare the identification performance of the algorithm to anything. We suffice to observe that a naive algorithm would rely on the use of identifying sensors, and could thus only identify an agent if it is sensed by an identifying sensor.

During training for the identification task, we turn the sensor in the top left corner of the environment into an identifying sensor. Then we let the clockwise and counter-clockwise Wall Hugger agents take turns in walking a circle, starting from the top left corner. During the actual experiment, we shall let the agents start in the bottom right corner of the environment, so that the algorithm will have to use the learned motion models to identify the agents.

4.3 Results

In Figure 4.2, the most important results for the tracking and prediction tasks are shown. Not shown in the figure is the result that in a noise-free environment, both the naive method, the untrained and the trained algorithm achieved perfect scores for all tracking experiments. Thus, for both the Random Goal agent and the Wall Hugger agent, the naive method, the trained and the untrained algorithm achieved a 100% score on both the Precision and the Recall performance measure. This is not very surprising, since in this noise-free environment, all necessary information to achieve these perfect scores is available.

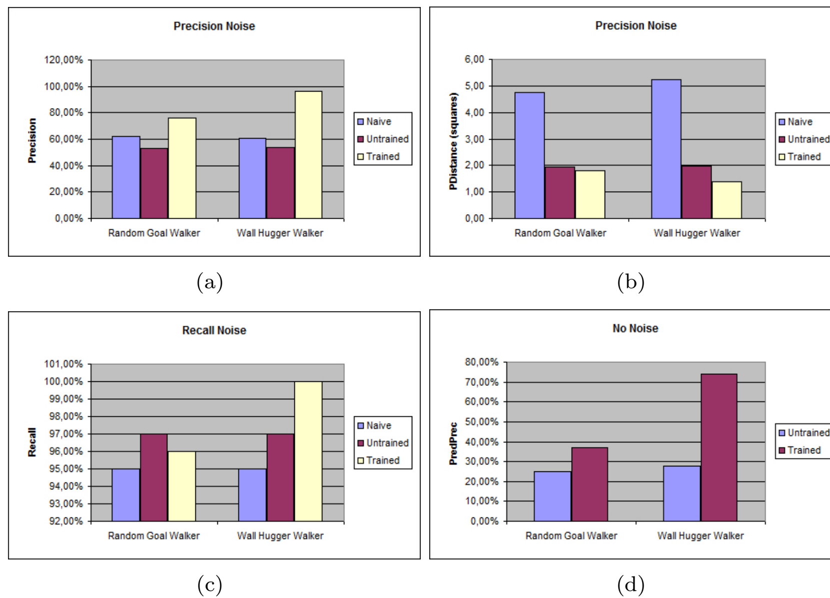


Figure 4.2: Results from the experiments with the simulation environment.

4.3.1 Tracking

From Figure 4.2(a) we can see that training the algorithm has a positive influence on the Precision score of the algorithm, especially for the Wall Hugger agent. That the effect is stronger for the Wall Hugger agent makes sense: since its behavior is more predictable it is also easier for the algorithm to distinguish false sensor measurements from correct ones. From the score of the naive method we can see that 40% of all sensors readings indicating the presence of an agent was false. For the Wall Hugger agent however, the trained algorithm was nearly able to completely nullify the effect of this noise.

We also see that the untrained algorithm was not capable at all of achieving a good Precision score when faced with P-noise. However, from Figure 4.2(b) we see that it at least acquired a much lower PDistance score than the naive method. This means that the erroneous spikes by output neurons were quite near the actual location of the agent. This makes sense: near the actual location of the agent, a lot of activity already enters the network. An erroneous sensor reading in the vicinity of the agent could then cause many output neurons in

the vicinity of the agent to spike. In contrast, an erroneous sensor reading at a location where little activity has entered the network will likely not cause any spiking by output neurons. Thus, the untrained algorithm is capable of filtering the extremely wrong sensor readings, and allows for tracking with room-level accuracy.

From Figure 4.2(c) we can see that in the presence of R-noise the algorithm is not able to achieve a good Recall score for the Random Goal agent in either trained or untrained form, compared to the baseline performance. This is explainable. Since the behavior of the Random Goal agent is random to a high degree, sensor readings simply are required to be able to correctly determine the location of the agent. We see that in the case of the Wall Hugger agent, the trained algorithm has learned the motion model of the agent, and thus does not require all sensors readings any more to be able to determine the location of the agent.

4.3.2 Prediction

Finally, in Figure 4.2(d) we can see the performance of the algorithm in trained and untrained form in a noise-free environment for the Wall Hugger agent and the Random Goal agent. As with the experiments on Recall, we can see that it is hard for the trained algorithm to achieve a good score for the Random Goal agent. This has the same cause: the motion model of the Random Goal agent is random to a high degree, making it hard to predict its movements based only on this motion model. A solution might be to base predictions also on the current direction of the agent, as is done in many tracking algorithms, as described in chapter 1. How to implement this is beyond the scope of this thesis.

We can see that for the Wall Hugger agent, predicting the movements based on the motion model is much easier. Again, however, a higher score could be achieved by also using the direction an agent is moving in. This is because the agent walks the central corridor in both directions. As the algorithm only learn this fact, it is incapable of predicting correctly when the agent is in the central corridor.

4.3.3 Identification

As mentioned earlier, for the identification task no quantitative data were gathered. Rather, we will analyze if the algorithm is capable of identifying agents in an experiment. In Figure 4.3, we see part of the experiment, where the counter-clockwise Wall Hugger agent has just started a fresh round. In the figure, the top part is the actual path of the agent, the bottom part is the output of the algorithm. In this output, every square is divided in two, each half representing the activation for one identity of the output neuron on that square. The left half is for the counter-clockwise Wall Hugger agent, the right half for the clockwise agent. The darker the square, the higher the activity, and a completely black square means the output neuron spikes. We have zoomed in on only the bottom right chamber and part of the corridor.

As one can see, the first time step the algorithm does not output anything significant yet. The activation of the output neuron at the location of the agent is highest, but the identity is unknown. One can see, however, that the algorithm already predicts that if the agent is the counter-clockwise agent it will go up,

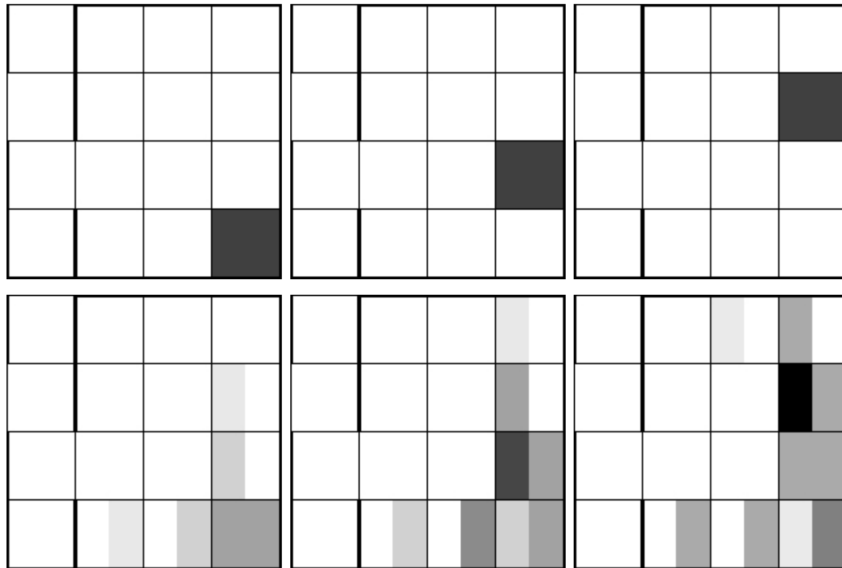


Figure 4.3: Top: the counter clockwise Wall Hugger agent takes the first three steps of a fresh round in the environment. Bottom: the output the algorithm delivers.

and that if it is the clockwise agent it will go left. During the second and third step we can see the algorithm gathers conclusive evidence that it is indeed the counter-clockwise agent. In Figure 4.4 the first three steps of a fresh round for the clockwise agent are shown, and one can observe that the algorithm identifies the agent in the same way.

Thus, the algorithm is indeed capable of identifying agents using their motion model. The algorithm stores these motion models distributed over the weights of connections. Every weight of a connection signifies whether the corresponding class of agent travels from the location of the input neuron to the location of the output neuron or not. Thus, the algorithm uses only the paths that agents walk to identify them. In the central corridor this would lead to problems if we had not introduced the procedure for the propagation of identities. This is because both classes of agents we used walk the corridor in both directions.

Though we have now achieved significant identification results using only the paths that agents walk, this is still only a preliminary result. The algorithm currently is incapable of distinguishing agents that walk the same path but at different speeds, or at different times. Further research could be directed at incorporating such information.

4.4 Discussion

The algorithm we have proposed is able to successfully track agents in a noise free environment. It is also capable of predicting the movements of agents, if the motion model of the agent does not contain too many random elements. We have conjectured that if the algorithm would somehow take into account the current direction of motion of an agent, its prediction performance would

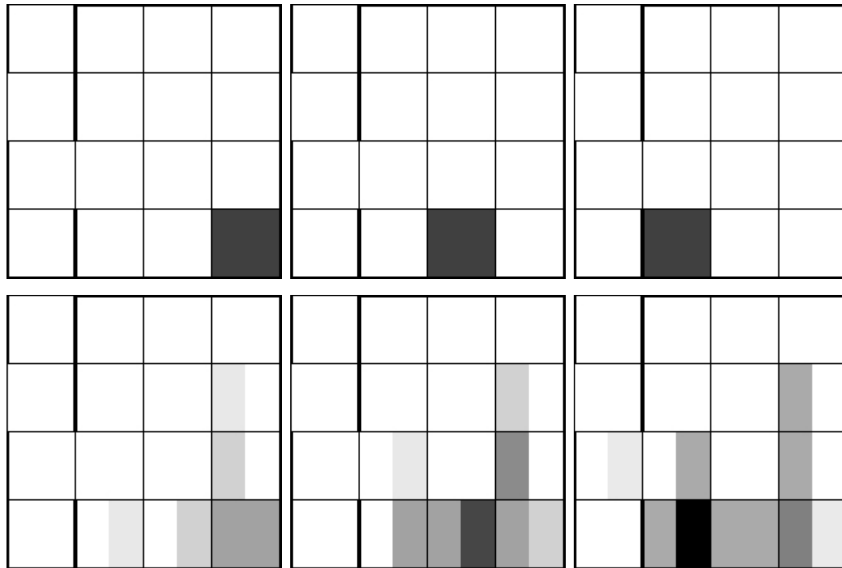


Figure 4.4: Top: the clockwise Wall Hugger agent takes the first three steps of a fresh round in the environment. Bottom: the output the algorithm delivers.

go up. The algorithm is capable of learning the motion model of agents, and then using the learned motion models to identify agents. The motion models the algorithm learns currently are far from complete (they do not contain information regarding speed of agents or the time they walk certain paths), and we theorize that by expanding the learned motion models, the algorithm could learn to identify better.

We have also tested whether indeed the algorithm is capable of dealing with noise in the environment. We have seen that introducing a variable θ allows the algorithm to deal with both P-noise and R-noise. However, the procedure we proposed for nodes to determine θ may not be feasible in a real application. We proposed that output neurons whose activation was higher than a low base value of θ, would exchange their activation to determine which neuron had the highest activation. In practice, this would require a lot of communication between nodes, and this is not desirable, for reasons outlined in chapter 1.

We have also observed that using θ, a trade off can be made between Precision and Recall. A higher θ would increase Precision, but lower Recall, and a lower θ would raise Recall and lower Precision. We claim that our algorithm would do well for all practical purposes if θ is fixed at a low value, so that a high Recall score is achieved, at the expense of Precision. This is because, as we have observed, the erroneous spikes that cause a low Precision score are generally in the vicinity of the agent. We have observed this from the PDistance scores the algorithm achieved. Thus, a low Precision score of the algorithm mainly means that the algorithm is not capable of tracking on a square meter scale anymore, but rather on a room-level scale. This will generally be enough for the tasks the system will eventually have to carry out.

Chapter 5

Validation using Prototype

The algorithm we have proposed is distributed and scalable by design, which makes it fit for implementation on a wireless sensor network. However, we have to verify whether the tasks an individual network node has to carry out for the algorithm are not too demanding for the network node. We have observed that in a wireless sensor network, the network nodes typically have only limited computational capabilities and can store only a limited amount of data. Moreover, its communicational abilities are also limited and costly in terms of power consumption.

In this chapter we shall explore the practical feasibility of the algorithm by creating a prototype of a sensor network implementing our algorithm. We will test the prototype in a toy-setting, and will not implement the identification part of the algorithm. The goal is not to test the performance of the algorithm in a real world scenario, but rather to find and discuss issues that arise when actually implementing the algorithm on a sensor network. We will first discuss the environment and hardware used for the prototype. We will then discuss how the algorithm was implemented on the network nodes. We will finish by discussing the prototype and additional issues encountered.

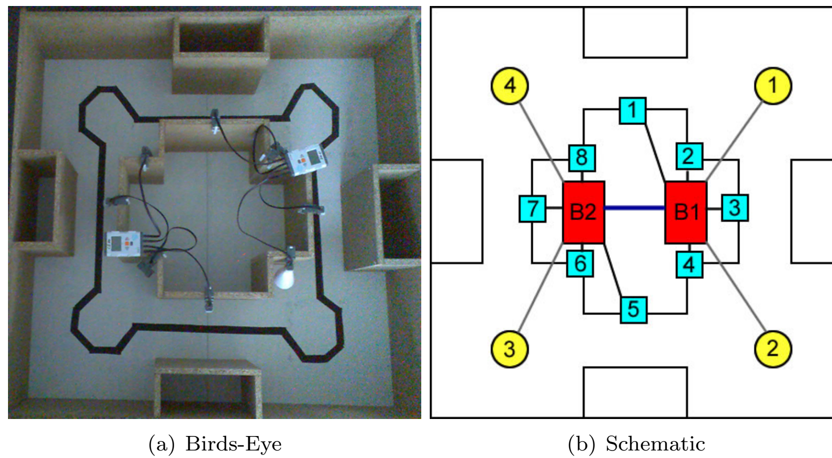
5.1 Prototype

In this section we will describe the prototype that we have created. We will first describe the environment and the agents used in the environment. We will then turn to the hardware and the actual sensor network we created.

5.1.1 Environment and Agent

Using chipboard we created a closed environment: a square box measuring 1m25 by 1m25, divided into four rooms. Each of these rooms is connected to two other rooms with a corridor. The floor of the environment is painted white, and with black tape a line is created on the floor of the environment. For a birds-eye picture of the environment see Figure 5.1(a).

As agent we use a small robot constructed using a LEGO Mindstorms NXT kit (see `http://mindstorms.lego.com` for more information on the Lego Mindstorms NXT kit, see Figure 5.2 for a picture of the robot agent.). The robot



(a) Birds-Eye (b) Schematic

Figure 5.1: A Birds-Eye and a Schematic overview of the Prototype environment.

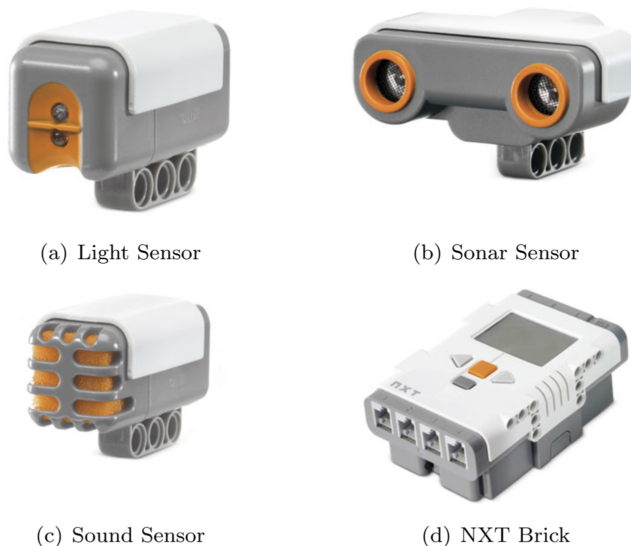
employs a light sensor to follow the black line on the floor of the environment. We covered the agent with black paper, so that it would be clearly separable from the rest of the environment by the various sensors we used in the sensor network. The agent continually moved around in the environment in circles.

Figure 5.2: The robot that serves as agent and follows the line on the floor in the environment.

5.1.2 Hardware

For the sensor network we used more LEGO Mindstorms NXT kits. Each of these kits contains, amongst others, one NXT brick, a sound sensor, a sonar sensor, a light sensor, a pressure sensor and three interactive servo motors (see Figure 5.3 for pictures of some of these components). The NXT brick is a computational device. It has a 32-bit ARM7 microprocessor and can be connected to up to four different sensors and up to three motors, and is equipped with a Bluetooth radio. With the Bluetooth radio a brick can in principle be connected to up to three other bricks, but it is not capable of broadcasting messages to these three bricks, let alone to all bricks in the vicinity.

Of the sensors we will not use the pressure sensor, since it is essentially a button, and we do not want to have the sensor network relying on an agent pushing buttons. The light sensor measures the light intensity directly in front of it. The sonar sensor uses ultrasonic waves to measure the distance to the nearest object, up to a distance of 255 cm. Using multiple sonar sensors close to each other may cause them to interfere, since one sensor may receive the waves of the other sensor. The sound sensor measures the sound level in the environment. However, to make it sensitive enough to soft noises, and to make it insensitive to background noise, we outfitted the sound sensors with a cone made of ordinary printing paper.



(a) Light Sensor (b) Sonar Sensor

(c) Sound Sensor (d) NXT Brick

Figure 5.3: Components of the Prototype Wireless Sensor Network.

5.1.3 Sensor Network

The use of the LEGO Mindstorms NXT kits is far from optimal for a wireless sensor network. As wireless sensor nodes, individual bricks are not cheap (€159.99 for a single brick, €299.00 for an entire kit), and they are incapable of broadcasting messages. However, they were readily available, and programming them is relatively easy, as many programming languages with communities already exist [43]. Thus the choice for using this hardware was made, though it did require to somewhat alter the original idea of equipping every node with one sensor and having every node implement one input neuron and one output neuron.

The specific programming language we used to program the bricks, nxtOsek (see `http://lejos-osek.sourceforge.net/`), only supports two bricks at a time to be connected. Moreover, this connection can not be set up or broken down at runtime. Thus, our sensor network is limited to two bricks. To each of these two bricks, we connect a sonar sensor, a sound sensor, and two light sensors. The sound and sonar sensors are each placed in a separate room, while the light sensors are placed in the corridors.

Thus we let each brick implement four input neurons, one for every sensor, and two output neurons. We initialize the algorithm in such a way that the output neurons are simulated to be located in a room. Though every output neuron in principle is connected to every input neuron, since the two bricks that together implement all input and output neurons are connected, we artificially define a different connectivity. This is to make the prototype more interesting and to make it somewhat more realistic. We define each output neuron to be connected to the five input neurons that are nearest to it in terms of the distance the robot has to travel from the location of the output neuron to the location of the input neuron. See Figure 5.1(b) for a schematic overview of the environment containing the implemented input and output neurons.

5.2 Algorithm

For the implementation of the algorithm, besides the parameter settings, an important issue needs to be discussed: how to have individual nodes perform the right action at the right moment.

5.2.1 Software Architecture

In the algorithm we proposed, three different tasks can be discerned that individual network nodes must perform. First, the nodes must use their sensor to observe their environment, set the activations of input neurons accordingly, and send messages to other nodes. Second, they must update the activations of their output neurons, update weights, and send messages to other nodes if the output neuron spikes. Third, nodes must constantly be able to receive incoming messages.

Recall that we defined the algorithm to work in discrete time steps. During a time step, a node would set the activations of its input neuron, update the activations of the output neuron, apply the learning procedure and send messages. Implementing this would be trivial, but two challenges arise. We are aiming at creating a system with no central component that coordinates when nodes perform which task. Thus, the network will not be synchronized, and nodes can not be sure when they will be receiving messages from other nodes. This means that nodes must be able to receive a message at all times.

The second challenge is that taking one sensor reading every time step may not be enough to reliably detect an agent if it is present. If we define individual time steps to be one second, an agent may pass a sensor between two readings, and thus go unnoticed. Thus one would rather have the node take multiple sensor readings per time step. This also allows us to make sensors more robust to noise, as we can average over multiple sensor readings to determine the activations of input neurons.

The nxtOsek programming language we used to program the bricks is based on the Osek Real Time Operating System. Real Time Operating Systems (RTOS) are designed to create applications that must work in real time. The general focus is on getting tasks done before certain deadlines, rather than getting as much done as possible. Thus, nxtOsek allows for the creation of tasks: a set of instructions that is executed at fixed intervals. For each task, this interval

can be defined, and the RTOS takes care of executing every task at the right time.

We use this feature by creating three different tasks giving each of the three tasks defined above a separate task in the application. Thus we can in a natural way allow nodes to be constantly listening for incoming messages, while taking sensor readings and updating the activations of output neurons on a less frequent basis.

One downside of the nxtOsek programming language, is that it does not allow for the dynamic allocation of memory. Since nodes have to store incoming messages for some time, this is somewhat problematic. We solved this by beforehand allocating a fixed amount of memory to the storing of messages. Messages were deleted if they were the oldest message in memory and a new message arrived, or by the constraint we had already introduced in chapter 3: after $\frac{\log \tau}{\log \gamma}$ time steps.

In the prototype, we set the size of time steps to one second. In algorithm 5.1, the pseudo code for the algorithm as implemented on the bricks is depicted. Note that every brick implemented four input neurons and two output neurons. We achieved this by adding a loop over all input and/or output neurons in every task, but we have omitted these loops to keep the algorithm more general.

Programming nodes using an RTOS thus makes dealing with the timing of the execution of tasks a straightforward business. If a programming language based on an RTOS is not available, the timing of the execution of tasks is less straightforward, but not impossible. Instead of defining separate tasks and defining for each task an interval at which it must be executed, one could could create a loop in which tasks are executed if it is their time. In algorithm 5.2 example pseudo-code is depicted for this situation. It assumes that the node has a system clock available with millisecond precision.

5.2.2 Parameter Settings

As mentioned, we set the size of time steps to one second. We set $\gamma = 0.8$, $\theta = 4.5$, $\sigma = 1$ and $\omega = 0.25$. We define the δ values for every pair of connected input and output neuron as defined in table 5.1, in the first column of every output neuron. Though we have not used the usual procedure for setting the δ values, as defined in equation 3.3, these δ values do approximate the time it takes the agent to travel from the location of the sensor to the simulated location of the output neuron.

5.3 Results and Discussion

After installing the entire sensor network in the environment, the agent was set loose in the environment. The network was able to satisfactory track the agent. Moreover, as depicted in table 5.1, after the system ran for some time, a number of weights went from a value of 1 to a value of 0. Comparing the table of learned weights to the schematic picture of the environment, we see that the network has learned that the agent that was present in the environment always moved in counter clockwise circles.

Though this is an important results, we can note two algorithm specific points that need further investigation. While developing the algorithm, we have

Algorithm 5.1 Algorithm using Tasks

```
1: TASK: Sensor Sampler
2: EXECUTE: 100ms
3: S ← SensorReading
4: if Counter = 10 then
5:    Set A_j^self
6:    if A_j^self ≥ 1 then
7:       Broadcast A_j^self
8:       Put A_j^self in StoredMessages_1
9:    end if
10: end if
11: Counter = Counter mod 10
12: Counter + +
13:
14: TASK: Activity Updater
15: EXECUTE: 1000ms
16: Calculate A_i
17: if A_i^self ≥ θ then
18:    for all A_j ∈ StoredMessages_1 do
19:       Strengthen w_ij
20:       Put A_j in StoredMessages_2
21:    end for
22:    for all A_j ∈ StoredMessages_2 do
23:       if t(A_j) + δ_ij + log τ / log γ < t(Current) then
24:          Remove A_j from StoredMessages_2
25:       end if
26:    end for
27: else {A_i^self < θ}
28:    for all A_j ∈ StoredMessages_1 do
29:       if t(A_j) + δ_ij + log τ / log γ < t(Current) then
30:          Weaken w_ij
31:          Remove A_j from StoredMessages_1
32:       end if
33:    end for
34: end if
35:
36: TASK: Message Listener
37: EXECUTE: 50ms
38: if Message in Buffer then
39:    Extract A_j from message
40:    Put A_j in StoredMessages_1
41:    Remove message from Buffer
42: end if
```

Algorithm 5.2 Algorithm making use of a giant loop

1: **loop**
2: $CurrentTime \leftarrow SystemClockTime$
3: **if** ($CurrentTime$ mod 100) $> Counter1$ **then**
4: EXECUTE Sensor Sampler
5: $Counter1 + +$
6: **end if**
7: **if** ($CurrentTime$ mod 1000) $> Counter2$ **then**
8: EXECUTE Activation Updater
9: $Counter2 + +$
10: **end if**
11: **if** ($CurrentTime$ mod 50) $> Counter3$ **then**
12: EXECUTE Message Listener
13: $Counter3 + +$
14: **end if**
15: **end loop**
16:
17: METHOD: Sensor Sampler
18: $S \leftarrow SensorReading$
19: **if** $Counter = 10$ **then**
20: Set A_j^{self}
21: **if** $A_j^{self} \geq 1$ **then**
22: Broadcast A_j^{self}
23: Put A_j^{self} in $StoredMessages_1$
24: **end if**
25: **end if**
26: $Counter = Counter$ mod 10
27: $Counter + +$
28:
29: METHOD: Activation Updater
30: Calculate A_i
31: **if** $A_i^{self} \geq \theta$ **then**
32: **for all** $A_j \in StoredMessages_1$ **do**
33: Strengthen w_{ij}
34: Put A_j in $StoredMessages_2$
35: **end for**
36: **for all** $A_j \in StoredMessages_2$ **do**
37: **if** $t(A_j) + \delta_{ij} + \frac{\log \tau}{\log \gamma} < t(Current)$ **then**
38: Remove A_j from $StoredMessages_2$
39: **end if**
40: **end for**
41: **else** $\{A_i^{self} < \theta\}$
42: **for all** $A_j \in StoredMessages_1$ **do**
43: **if** $t(A_j) + \delta_{ij} + \frac{\log \tau}{\log \gamma} < t(Current)$ **then**
44: Weaken w_{ij}
45: Remove A_j from $StoredMessages_1$
46: **end if**
47: **end for**
48: **end if**
49:
50: METHOD: Message Listener
51: **if** Message in Buffer **then**
52: Extract A_j from message
53: Put A_j in $StoredMessages_1$
54: Remove message from Buffer
55: **end if**

	Neuron 1			Neuron 2			Neuron 3			Neuron 4		
	δ	w_0	w_∞	δ	w_0	w_∞	δ	w_0	w_∞	δ	w_0	w_∞
Sensor 1	10	**1**	**0**							10	1	1
Sensor 2	0	1	1	25	**1**	**0**				25	1	1
Sensor 3	10	1	1	10	**1**	**0**						
Sensor 4	25	1	1	0	1	1	25	**1**	**0**			
Sensor 5				10	1	1	10	**1**	**0**			
Sensor 6				25	1	1	0	1	1	25	**1**	**0**
Sensor 7							10	1	1	10	**1**	**0**
Sensor 8	25	**1**	**0**				25	1	1	0	1	1

Table 5.1: Results from the Prototype experiment

assumed that only binary sensors were available. We defined binary sensors to be sensors that are capable of only outputting a 0 or a 1 for every sensor reading. However, none of the three types of sensors we used was truly binary. So long as using the sensors is feasible from a cost and energy consumption point of view this is not a problem. However, one does have to think carefully about how to use the data the sensors gather. Since we have as of yet no way of using non-binary sensor data in the algorithm, the data gathered by non-binary sensors will have to be preprocessed. And while for a binary sensor the interpretation of sensor values is straightforward (a 0 means no agent is detected, a 1 means an agent has been detected), for non-binary sensors this is not so. In our prototype we implemented a threshold on the value of each sensor readings, such that if the reading exceeded the threshold, we interpreted this as a detected agent. For the light sensor, this resulted in clouds suddenly blocking or unblocking the sun could cause the sensor to detect an agent, while none was present. Thus, we identify the preprocessing of raw sensor data as a point for further investigation.

The second point concerns the parameter settings of the algorithm. The parameters used in this prototype were found by trying multiple settings until a combination that worked was found. Though one can find reasonable settings by analyzing the environment and the way the parameters of the algorithm interact, no fixed procedure for doing this is available yet. It is highly undesirable to try multiple parameter settings in a real application of the sensor network as it would take a long time to reprogram all nodes multiple times. Thus, we also identify the development of a standard procedure for the algorithm to determine the optimal parameters at run time as a point for further research. The algorithm could detect specific features of the network, such as the node density, and use these to make an initial estimation of the correct parameter settings. It could then try to optimize the parameter settings at run time.

A less algorithm specific point that deserves attention is the use of memory. Depending on the specifications of the hardware used for the sensor network, it may be necessary to minimize the amount of memory the algorithm needs. One source of data that nodes have to store are the activations of input and output neurons and weights of connections. Thus far, we have assumed that weights and activations are stored as floating point values. However, with proper conversion, it is possible to store them as bytes. Thus every weight and activation would require only 8 bits to be stored, which depending on the size of floating point values on the specific platform chosen, can be a significant saving.

Chapter 6

Discussion and Conclusion

In this final chapter we will reflect on the work presented. We will discuss to what extend the algorithm fulfills the three tasks we identified: tracking, prediction and identification, and if the algorithm successfully deals with the challenges we identified. Subsequently, we will present a number of possible directions for future research. We will then conclude the thesis.

6.1 Tasks

Using five scenarios we identified three basic information gathering tasks that an ambient intelligence system for institutionalized health care must perform. These tasks were *tracking*, *prediction* and *identification*. For the tracking task, every node that is part of the system has to be able to determine, at all times, whether an agent is present at the location of the node in the environment. For the prediction task, nodes have to be able to determine, at all times, whether an agent is approaching the location of the node in the environment. For the identification task, nodes have to be able, if they determine that an agent is present at the location of the node in the environment, to also determine the class of that agent.

The aim of the thesis was to develop a preliminary proposal for an ambient intelligence system that can perform these three tasks. We have proposed to use a wireless sensor network, and view it as a physical artificial neural network. We have tested the algorithm for its performance on the three tasks using a simulation environment.

We saw that on the tracking task the algorithm achieved a perfect score in a noise-free environment. In a noisy environment, the algorithm did not achieve perfect scores for tracking agents on the level of individual nodes, but did allow for room-level tracking of agents. We have also identified the threshold θ for output neurons to spike as a factor in the amount and type of noise the algorithm can deal with.

For the prediction task, we proposed that the algorithm learns the typical behavior of agents, and uses it to predict where agents are going. We saw that before learning the algorithm generates a lot of wrong predictions, but that, depending on the amount of randomness in the typical behavior of agents, this number of wrong predictions could be significantly limited. A perfect score has

not been achieved yet though, and thus improvements are needed. We will give directions on this in the further research section.

For the identification task, we let the algorithm learn the typical behavior of different classes of agents. By comparing the behavior an agent exhibits to the typical behaviors, the class of the agent could be determined. We saw that this worked quite well. However, to allow the network to learn the typical behavior of classes of agents, we were forced to temporarily introduce at least one identifying sensor. We have argued that the reason this was necessary was a fundamental one, that could not be circumvented. In that light our solution provided for a minimal intrusion on the privacy of agents.

Thus, we can say we have succeeded at creating a preliminary proposal for an ambient intelligence system that performs the three tasks. The performance of the algorithm is not yet optimal, and so improvements are needed. However, as we will discuss in a bit, we have ideas on how to improve the performance of the algorithm. Seeing how the approach we took was quite new, we can say that the results are at least promising.

6.2 Challenges

We made two important design decisions that caused a number of challenges to arise. First, we decided to use a wireless sensor network and second, we decided to use only binary and anonymous sensors. We will now discuss whether the identified challenges were met.

The first challenge we identified was *energy consumption*. Nodes in a wireless sensor network are battery operated, and thus must use this limited source of power in an economical way. In this thesis, we have not extensively treated this issue, or how our algorithm deals with it, and this is an important point for future research. However, we can say some things about this point. It is generally accepted that for a wireless sensor node, communication is the operation that consumes the most energy. Our algorithm only requires the communication of data to nodes in the vicinity, and messages that need to be send are never larger than a few bytes. Thus, we conjecture that the amount of energy used on communicating will be limited.

The second challenge we identified was that any software architecture developed for a wireless sensor network must be *distributed*. We adopted the paradigm of artificial neural networks, and viewed the entire sensor network as a neural network. By defining only limited connectivity (i.e. the connectivity of the neural network as equal to the connectivity of the sensor network) we have inherited the property of distributedness from neural networks.

The third challenge was that a wireless sensor network needs to be *self-organizing*. Nodes should not have to be programmed individually with data that is specific to that node, such as data on the physical location of the node in the environment, or application specific data. Our algorithm does not depend on any node-specific information. Rather, we assume that nodes can identify neighboring nodes and determine the distance to these nodes. We have given ideas as to how this could be done. Our algorithm also does not require any application specific data to be present at the nodes. Rather, nodes learn application specific data, such as the typical behavior of agents, themselves.

The last challenge was that the algorithm would have to be robust to noise.

In the simulation we tested the performance of the algorithm in a noisy environment. While the scores of the algorithm were not as good as in a noise-free environment, we saw that the algorithm still performed reasonably well, and at least allowed for room-lever tracking of agents. However, to test the performance of the algorithm under the presence of real noise, the algorithm will have to be tested in a real world environment that is less limited than the one presented in chapter 5.

6.3 Further Research

As mentioned already, work on this algorithm is not finished, and thus multiple issues for further research can be identified. These issues can roughly be distinguished into two groups: issues pertaining to the algorithm itself, and issues that go beyond the algorithm.

6.3.1 Algorithm Specific

The first issue concerns the assumption that agents move about in the environment at a fixed speed, and that this speed is known. This is a limiting assumption, since agents typically do *not* walk at a constant speed. Moreover, the typical speed at which agents move may be used to identify agents: elderly people typically walk slower than, for instance, nurses. Thus, it would be desirable to augment the algorithm such that it can deal with variable move speeds, and can use these to identify agents.

One way to accomplish this would be to make the delay factors between neurons dynamic. Recall that a delay factor essentially states how long an agent is expected to take to travel from one neuron to another neuron. Thus, delay factors are directly related to the move speed of agents. By using fixed delays, we can only work with fixed move speeds. We could, however adapt the learning procedure to also work on delays. Instead of only strengthening and weakening connections, we could let nodes change the delays to better match the time between receiving a message from an input neuron and arrival of the agent. By introducing separate delays for individual classes of agents, we can even use the typical move speeds of classes of agents to identify agents.

The second issue concerns the thresholds for neurons to spike. We have until now assumed that these thresholds are fixed, but we also observed from the simulations that a variable threshold is desirable. This was so that the algorithm can correctly deal with noise in the environment. Exactly how a dynamical threshold could be introduced in a meaningful and natural way is as of yet unclear, and this will have to be researched. We suggest to look for an answer again in the field of spiking neural networks, as dynamical thresholds have been suggested there also [24].

Third, we have thus far only tested the algorithm on an environment in which at most one agent was present at a time. In a real application, the algorithm will have to be able to deal with situations where multiple agents are present in the environment at one time, and possibly even close together. Thus, the algorithm must be tested on the task of tracking multiple agents at the same time. Possibly, the algorithm will require more work to be able to deal with such situations.

The last algorithm specific issue is the issue of energy consumption. As we have stated, communication generally consumes more energy than computation, and thus we have attempted to keep the amount of data that needs to be communicated by nodes limited. However, we have not incorporated any mechanisms specifically for the conservation of energy. One example of such a mechanism is putting nodes into a sleep mode if nothing is happening, and waking them up if agents are coming. Energy consumption will have to be incorporated in the algorithm in a more prominent way.

6.3.2 Beyond the Algorithm

As we stated in the introduction, the aim of this thesis was to propose an ambient intelligence system that can perform three tasks: tracking, prediction and identification. We also stated that question of how to use this information in a meaningful way was left open. This is, thus, an obvious issue for further research. The information that the algorithm is able to gather now must still be used somehow for the system to perform the higher level tasks that were described in the scenarios in chapter 1. Thus, research will have to be conducted to how specific actuators could be integrated into the system, and how the system can relay information to human beings that are interested in the information.

The last issue for further research we identify here is to find or develop a suitable platform to implement the algorithm. In our prototype, we have implemented the algorithm on Lego Mindstorms NXT bricks, but we saw that this platform was far from ideal. To select or develop a suitable platform, it is necessary to construct a list containing all requirements that such a platform should meet. One could think, for instance, of the minimal computational power or storage capacity, but also specific communication protocols. One could imagine that for the normal operation of our algorithm all nodes must be able to broadcast messages, but that for reprogramming an entire network with a new program version or during a network initialization phase, a directed form of communication is required.

6.4 Conclusion

In this thesis, we have created a preliminary proposal for an ambient intelligence system for institutionalized elderly care. Specifically, we proposed an algorithm that can operate on a wireless sensor network. The algorithm performs three tasks that are essential for the system to perform high level tasks; tracking, prediction and identification. Moreover, the algorithm has some features that are essential for any application that is developed to operate on a wireless sensor network. The algorithm is distributed by design, self-organizing and quite robust to noise in the input. We tested the performance of the algorithm in a simulation environment and we made a first step towards a real implementation by creating a prototype.

The novelty of the work presented in this thesis in twofold. First, this is an attempt at creating an algorithm for tracking, prediction and identification that is distributed, self-organizing, robust to noise, energy efficient and that assumes only binary and anonymous sensors. As the field of wireless sensor networks is still relatively young, how to best achieve these features is still

subject of much research. Testimony to this are for instance the various projects concerning wireless sensor networks that we have already briefly discussed in the introduction. We feel that this work fits neatly in the field, because we have proposed an algorithm that has some of these features by its very nature.

The second point of novelty of this work concerns the use of artificial neural networks. To our best knowledge, this is the first attempt at applying the paradigm of artificial neural networks to the field of sensor networks. We believe that the combination of the two is a natural one: both neural and sensor networks consist of many small units of computation that are connected in some way and cooperate to achieve a task. We have seen that neural networks already possess many of the features that we identified to be requirements for applications for wireless sensor networks. In this thesis, we have shown how the two can be combined in a meaningful and straightforward manner.

Bibliography

[1] Horst Steg, Hartmut Strese, Claudia Loroff, Jrme Hull, and Sophie Schmidt. Europe is facing a demographic challenge, ambient assisted living offers solutions. Technical Report 004217, European Commission, March 2006.

[2] Roel Beetsma, Leon Bettendorf, and Peter Broer. The budgeting and economic consequences of ageing in the netherlands. *Economic Modelling*, 20:987–1013, 2003.

[3] Jari Ahola. Introduction: Ambient intelligence. *ERCIM News*, 47, October 2001.

[4] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J-C. Burgelman. Scenarios for ambient intelligence in 2010. Technical report, ISTAG, February 2001.

[5] Jürgen Nehmer, Martin Becker, Arthur Karshmer, and Rosemarie Lamm. Living assistance systems - an ambient intelligence approach -. In *Proceeding of the 28th international conference on Software engineering*, pages 43–50, 2006.

[6] Karen Zita Haigh, John Phelps, and Christopher W. Geib. An open architecture for assisting elder independence. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, 2002.

[7] Daniel H. Wilson. *Assistive Intelligent Environments for Automatic Health Monitoring*. PhD thesis, Robotics Institute Carnegie Mellon University, 2006.

[8] Thomas L. Harrington and Marcia K. Harrington. *Gerontechnology: Why and How*. Shaker Publishings B.V., Maastricht, Netherlands, 2000.

[9] Jan Gerrit Schuurman, Ferial Moelaert El-Hadidy, André Krom, and Bart Walhout. *Ambient Intelligence. Toekomst van de zorg of zorg van de toekomst*. Rathenau Instituut, Den Haad, Netherlands, 2007.

[10] Almende. Senzorg: Sensor netwerken in de zorg. Technical report, Almende B.V., 2007.

[11] Songhwai Oh, Luca Schenato, Phoebus Chen, and Shankar Sastry. A scalable real-time multiple-target tracking algorithm for sensor networks. Technical Report UCB//ERL M05/9, University of California, Berkeley, February 2005.

[12] H. Yang and B. Sikdar. A protocol for tracking mobile targets using sensor networks. In *Proceedings of the first IEEE International Workshop on Sensor Network Protocols and Applications*, 2003.

[13] P. Adema. Ambient living with embedded networks. Technical report, Development Laboratories (DevLab), 2007.

[14] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, January 2001.

[15] Michael J. Sailor and Jamie R. Link. "smart dust": nanostructured devices in a grain of sand. *ChemComm*, 2005.

[16] Wikipedia. Radio-frequency identification – wikipedia, the free encyclopedia, 2008. `http://en.wikipedia.org/w/index.php?title=Radio-frequency_identification&oldid=237031459` – [Online; accessed 9-September-2008].

[17] Michael Friedewald, Elena Vildjiounaite, Yves Punie, and David Wright. Privacy, identity and security in ambient intelligence: a scenario analysis. *Telematics and Informatics*, 24:15–29, 2007.

[18] Rahul Gupta and Samir R. Das. Tracking moving targets in a smart sensor network. *IEEE 58th Vehicular Technology Conference*, 5, 2003.

[19] Vincent S. Tseng and Kawuu W. Lin. Energy efficient strategies for object tracking in sensor networks: A data mining approach. *The Journal of Systems and Software*, 80:1678–1698, 2007.

[20] Javed Aslam, Zack Butler, Florin Constantin, Valentino Crespi, George Cybenko, and Daniela Rus. Tracking a moving object with a binary sensor network. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 150–161, 2003.

[21] Falko Dressler. Self-organization in sensor and actor networks. In *Almende Summer School*, 2008.

[22] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997.

[23] R. F. Thompson. *The Brain: A Neuroscience Primer*. W.H. Freeman, New York, 2nd edition, 1993.

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

[25] Donald Olding Hebb. *The Organization of Behaviour*. Lawrence Erlbaum Associates, 1949.

[26] L. F. Abbott and Sacha B. Nelson. Synaptic plasticity: Taming the beast. *Nature, Neuroscience*, 3:1178– 1183, 2000.

[27] Christian W. Eurich, Klaus Pawelzik, Udo Ernst, Andreas Thiel, Jack D. Cowan, and John G. Milton. Delay adaptation in the nervous system. *Neurocomputing*, 32-33:741–748, 2000.

[28] John A. Hertz, Richard G. Palmer, and Anders S. Krogh. *Introduction to the Theory of Neural Computation.* Addison-Wesley Publishing Company, Redwood City, CA, USA, 1991.

[29] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Massachusetts, USA, 1969.

[30] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[31] Teuvo Kohonen. The Self-Organizing Map (SOM), 2005. `http://www.cis.hut.fi/projects/somtoolbox/theory/somalgorithm.shtml` – [Online; accessed 18-July-2008].

[32] Eugene M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18, 2006.

[33] Jilles Vreeken. On real-world temporal pattern recognition using liquid state machines. Master's thesis, Intelligent Systems Group, Utrecht University, 2004.

[34] H. Jaap van den Herik and Eric O. Postma. *Future Directions for Intelligent Systems and Information Sciences*, chapter 7 - Discovering the Visual Signature of Painters. Springer, 2000.

[35] Andrzej Ksinsky and Filip Ponulak. Comparison of supervised learning methods for spike time coding in spiking neural networks. *International Journal for Applied Mathematics and Computer Sciences*, 16(1), 2006.

[36] Sen Song, Kenneth D. Miller, and L. F. Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature, Neuroscience*, 3, 2000.

[37] Tom Vercauteren, Dong Guo, and Xiaodong Wang. Joint multiple target tracking and classification in collaborative sensor networks. *IEEE Journal on Selected Areas in Communications*, 23(4):714–723, 2005.

[38] Chilukuri K. Mohan, Kishan G. Mehrotra, Pramod K. Varschney, and Jie Yang. Temporal uncertainty reasoning networks for evidence fusion with applications to object detection and tracking. *Information Fusion*, 8, 2007.

[39] Cristopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[40] Sebastiaan de Vlaam. Object tracking in a multi-sensor network. Master's thesis, Delft University of Technology, 2004.

[41] Dan Li, Kerry D. Wong, Yu H. Hu, and Akbar M. Sayeed. Detection, classification and tracking of targets in ditributed sensor networks. *IEEE Signal Processing Magazine*, 19(2):17–29, 2002.

[42] Amit Singhal. Modern information retrieval: A brief overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 24(4):35–43, 2001.

[43] Steve Hassenplug. Nxt programming software, 2008. `http://www.teamhassenplug.org/NXT/NXTSoftware.html` – [Online; accessed 4-October-2008].