Rantbot

Magnus Johansson

June 1, 2008
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Michael Minock
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

This Master's Thesis investigates a method for creating a rantbot, that is a chatterbot that goes on about a certain subject. The method involves using a recurrent artificial neural network where the nodes contain the sentences that the bot should spit out. The ranting behaviour was to be simulated by having activity flow through the network and thereby making nodes activate in a sequence. With this type of approach, the question was posed whether the effect of priming would be noticeable, that is would the chatterbot be able to tie one part of its output to other parts that it has generated earlier. In order to answer this question, experimentation was necessary and that involved creating an application that could produce this ranting output.

The program that was written for training and testing the chatterbot involved algorithms based on well known theories and equations for artificial neural networks. The program consists of a user interface in which the user can create a network and then run calculations to see the nodes that fire, and rate the links between the nodes. A number of tests were done using this application, on a variety of networks which tested different aspects of the application. Results showed that there are cases, most notably sequential networks, where the method works very well, and other cases where it's recommended to use other methods. The trainer application created was not well suited for the larger networks where the training time is excessive, since the user would become frustrated about the repetitive nature of the task.

Although the goal of creating a ranting chatterbot wasn't reached, some valuable insight was gained about the problems posed when training neural networks using the method investigated in the thesis. A few possible experiments were suggested for future investigations, as well as some improvements for the trainer application.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter aims to give an introduction to the problem, the task to be completed and finally some background on other work done within the field. Chapter 2 gives a description of the methods and mathematics used to accomplish the goals, chapter 3 describes the implementation process in more detail. In chapter 4 the experiments run using the implemented application are documented and commented on, and finally chapter 5 brings up some conclusions drawn from the work such as the usefulness of the method, possible improvements and future work.

1.1 Problem Statement

The objective of this thesis was to investigate a method for creating a program that could simulate ranting, which is a one-sided conversation about a specific topic. Often the rant is in a negative tone and the topic is of a political nature, for example when a person is unsatisfied with the current government and expresses his or her feelings by going on a rant where the concerns are brought up. The following quote is taken from a rant on a website devoted entirely to ranting, where people can submit their own rants and have them published on the website [2]. The author of the rant is frustrated with the British government and expresses his/her feelings by writing them down and putting it up for others to read.

> "They sit in the houses of parliament picking up obscene amounts of cash taking obscene backhanders and achieve nothing.Scandal after scandal and still they won't do the honourable thing and resign."

The method investigated in this thesis uses an artificial neural network where the nodes contain different sentences used to build up the rant, and these sentences get output in an order that makes sense to a human being. A brief introduction to neural networks can be found in section 2.1. One effect that is hoped to be achieved by using the method is that of priming. Priming is a form of implicit memory where for example a word mentioned to a person at one point in time can subconsciously help the person identify the same word at later point in time [1]. For example, let's say that person A and person B are having a conversation. Person A tells user B that he drives a Volvo, listens to ABBA and buys his furniture at IKEA. A few minutes later, person A asks person B to name a country. The probability is then pretty high that person B will respond with Sweden. The lack of this priming effect can seriously break the illusion of talking to a

human as it will make the bot seem to have an unnatural unawareness of what has been said. If this priming effect is achieved, a sentence A in the rant will lead to another related sentence B several sentences later in the chain on repeated runs.

For demonstrational purpose, the method can be implemented in a chatterbot. This chatterbot could output its results, the rants, to a chat room for example, and the reactions of the other participants in the chat room could be studied in order to see if the rant has any effect or if its just disregarded as being nonsense. Since the rant is one-sided, the work will focus primarily on generating the output of the rantbot and not on the parsing of input from a user. That being said, there will be a way for the user to trigger a sentence contained in the network which will make the bot start its rant.

1.2 Goals

The ultimate goal of the project was to have a working chatterbot which would receive text input from a user, and if the chatterbot's neural network contained a node related to the input, it would produce a rant on the same topic. For example, if the user input a sentence containing the word 'soccer' and if the data set of the chatterbot contained a category 'sports' with 'soccer' inside it, the bot would start saying something about soccer and then continue with other sentences relating to soccer or other sports. Since this goal was quite ambitious, some sub goals were created so as to be able to plan the time better.

- Sub goal 1 - implement a system for creating and manipulating artificial neural networks. This includes implementing a way to train the network in order to get satisfactory results.
- Sub goal 2 - connect the system to a database for storing and retrieving node data.
- Sub goal 3 - see how well the system handles different input and draw some conclusions about the usefulness of it.
- Sub goal 4 - create a chatterbot that can rant.

The first and third sub goals were the most important ones, while the second and fourth ones were of lower priority to complete if time permitted.

1.3 Related Work

Chatterbots deal with the problem of natural language processing. The first chatterbot, ELIZA [9], uses keywords along with decomposition rules to parse input and generates sentences with the help of reassembly rules. The best example from this bot is the psychotherapist script where the user inputs a statement and the bot asks a question according to the keywords in the sentence.

ALICE is a bot that was inspired by ELIZA [8], and uses pattern matching along with a database to create responses to the user's input. ALICE has thrice won the Loebner prize [5], a competition where judges rule which computer is the most human-like when conversing with a person. The competition is based on the Turing Test, named after Alan Turing who in 1950 suggested that for a computer to be considered intelligent, it would have to pass a test which involved interacting with a human being [7]. A person would write a series of questions for the computer to answer, and if the person couldn't

distinguish the responses written by the computer from those written by a human, the computer would pass the test. The first price in the Loebner contest has yet to be claimed, since none of the entries as of yet have passed the test, only come fairly close.

While these types of systems can do well at presenting output that closely resembles human conversation, a closer look will reveal that they don't really seem to understand the context of what they are talking about. They just fetch information from a database and construct neat sentences.

Milstein and Sandberg investigate semantics in natural language processing [3]. They use a semantic network, which is constructed as a recurrent network and is used to represent the connections between words, which are the nodes, in sentences. The network is divided into semantic groups, where the head node of the group has child nodes that represent words. New nodes are inserted when new input is seen, and the new node is then activated and its neighbour fires as well. This sets off a chain reaction of firing nodes that makes up a sequence of words that follow each other.

Chapter 2

Approach

In this chapter, the methods and mathematics to be used are investigated. First a brief introduction to artificial neural networks and the training of these is given. Next comes a description of the type of network used in this thesis as well as how calculations will be done on it. Finally, the mathematics to be used in the implementation are looked into and explained.

2.1 Artificial neural networks

Neural networks are modeled after the way neurons in the brain are connected to each other and the calculations done on them are based on how the neurons communicate with each other [7]. The neurons, or units, are connected in a network through links which are used to transport activation from one node to another. Neural networks can be constructed in many ways, most of which use a layered approach. There are single layer networks that consist of input units and output units only, see figure 2.1, then there are multilayer networks that can have one or several layers of hidden nodes between the input and output layers as shown in figure 2.2. The network is fed input into the input layer and this input propagates through eventual hidden layers down to the output layer which produces a result. The purpose of a neural network is that for some input, it should produce a certain output and the network can be trained until it produces that certain output. The flow of activity through a network is determined by

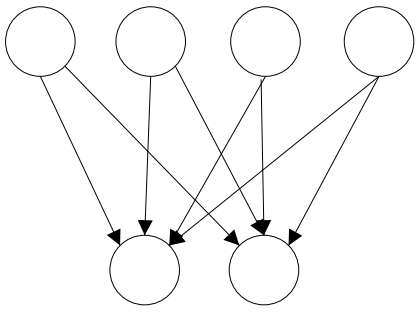

Figure 2.1: Single layer neural network.

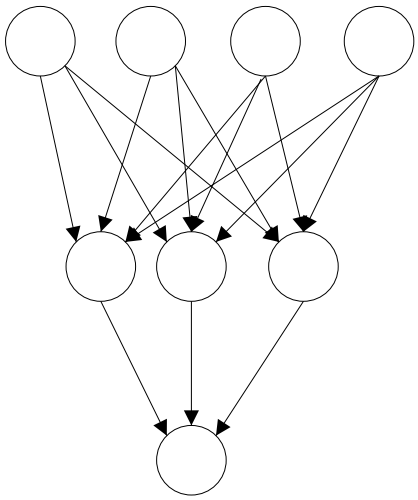

Figure 2.2: Multilayer neural network.

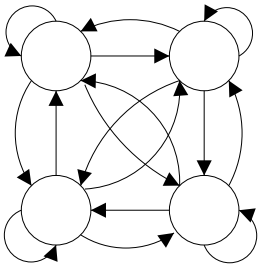

Figure 2.3: Fully recurrent neural network.

the connections between the nodes, where each connection has a certain weight to determine how much influence one node has on the node in the other end of the connection.

The connections between nodes can be one-way, as in feed-forward networks, but there's a special class of networks called Recurrent neural networks where the connections can go in both directions. The dynamic behaviour created by these types of networks allow them to support short-term memory. A subset of these recurrent networks are the fully recurrent neural networks [4]. In these networks there is a link between each and every node, and each node can act both as input node and as output node at the same time. The network in figure 2.3 is a fully recurrent network with four nodes, and so it's possible for every node to have a direct affect on every other node. This property makes it more intuitive to understand and train the network since one can directly change the relationship between any two nodes by just adjusting the weight of the link between them. For the ranting part, a Markov Decision process, or MDP, could be used for creating randomness. An MDP deals with states and defining policies for the transitions between the states [7]. The states in the case of the rantbot could be represented by the sentences, like the nodes in the neural network. For this, each jump from one node to

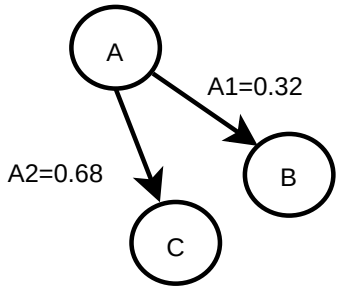

Figure 2.4: State A links to states B and C.

another would be decided by chance. Each out-link from a node would have a fraction number assigned to it, and the sum of all fractions should be 1. For example, if out-link A1 from node A to node B and has the value 0.32 and A2 from A to C has the value 0.68, the chance of node B being the next one in the chain would be 32%. Similarly, the chance of node C being the next one is 68%, see figure 2.4 for an illustration. This differs a lot from the idea of having weights and activity levels where it's the input links of the nodes that count. The problem with using this approach is that previous time steps are not considered. In neural networks, the activity of one node can affect the activity level of another node several steps in the future whereas with the MDP approach each "node" will just calculate a new random number and go with the matching output link.

2.2 Training neural networks

Training neural networks is done by adjusting the weights on the links until the desired output is produced, something which can be done in a number of ways. It can be done manually or automatically, and when it comes to systems that are to be trained as they are used it can be good to use supervised learning, allowing the users of the system that the network resides in to give feedback on how well the system is performing as it outputs its results. For example, figure 2.5 shows a process where the system is a search engine and the user types in a word to search for. The system then presents a list of results, as shown in figure 2.5a. The role of the search word would be that it represents a node in the network which has is directly connected to each of the nodes representing a result. The user can give feedback by telling the system how relevant each result is, thereby strengthening or weakening the connection between the search word and the result. Figure 2.5b illustrates this, and as seen in 2.5c the system could then sort the results based on their relevance to the search word.

Another way to train the network could be through reinforcement learning. Reinforcement learning deals with the process of having the network learn through a reward and punishment system [7]. Unlike supervised learning, this process doesn't need to involve any teacher which gives the network feedback as it learns. As an example of reinforcement learning, a child that touches a hot plate on a stove will feel pain and learn that it's a bad thing to do. The child learnt without a parent telling it not to touch a hot plate. For a ranting network, the end result is the full rant. For the rant to be considered good, the various sentences that makes up the rant should follow each

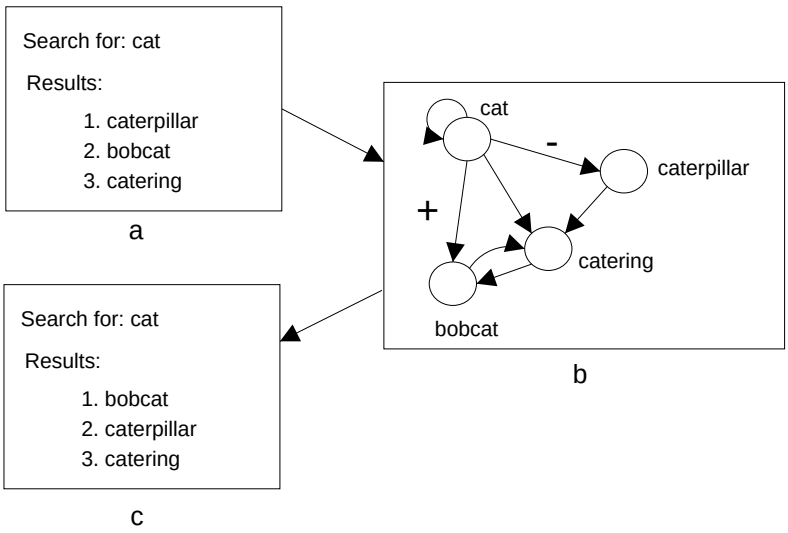

Figure 2.5: Example of a system using a recurrent neural network.

other in a way that makes sense. A bunch of sentences that are put together completely at random would make a bad rant. With reinforcement learning, the agent would be told only after the rant is complete if it makes sense or not.

Reinforcement learning can be used with both the MDP approach as well as the NN approach. In the case of MDPs, the agent can try out different probability values for the links and find the ones that lead to good ranting. For neural networks the weights of the links would get altered until the results are good enough. Working with reinforcement learning is a lot more complicated than working with supervised learning, so that approach will not be investigated in this thesis.

2.3 Methods

For the method investigated in this thesis, a neural network was to be used for storing and generating the data and the network was to be trained using supervised learning. The network is constructed such that each node is connected to all other nodes as well as itself. When the network is created, the weights of the links are set randomly using a uniform distribution of values within a certain interval. There are no discernible layers in this network, but instead each node acts both as an input unit and an output unit. The idea is that every possible input is represented by a node in the network, and when that input is to be fed into the network, the corresponding node is activated (setting its activity level so that it's above the threshold value). The calculations will then be run over the network to see whether any more nodes activate. Any nodes that do activate represent the output of the network, and calculations can be run again to see if any more nodes fire/activate. Having nodes popping like this is the key to getting a ranting behaviour, and the network must be trained to properly generate sequences that make sense, instead of creating some random slur.

The network is trained using supervised learning, so the user will tell the network whether its produced output is correct or not as it runs. When a node A fires after another node B, the user can rate the link between these two nodes. If the user rates the link negatively, the first node will have less influence on the second node. This in turn means that the next time node A fires, B will not follow if the link between them has become weak enough, that is if there is some other link $A \rightarrow C$ that has a stronger connection or if the activity level of B doesn't reach the threshold value. Input to the system is done through letting the user touch a node, meaning that the user can select a node and activate it, thereby raising its activity level above the threshold value. When the user has touched a node, calculations are run one the network and eventually a node will fire. If one does, calculations are run again to see if another one activates. This is repeated until no more nodes fire, and this popping behaviour is the base for the ranting part.

2.4 Network activity

The mathematics behind the equations used in the calculations on the network were based on well-known and tried methods used for general neural network calculations, as described by Hertz, Krogh and Palmer in their book [4]. There are three parameters involved when calculating the activity of the nodes in the network. The following list describes them briefly, and they are explained in more detail when used in the equations below.

- α The sum of activity in the network after one time-step.
- β Attention level, determines the spread of activity over the nodes in the network.
- δ The refractory time is the number of time-steps a node has to wait after it has fired before being able to fire again.

The activity of the nodes in the network is calculated separately for each node and then the values are normalised so they sum up to α. For each node, its activity at the next time-step, before normalisation, is calculated as follows.

$$a_i(t+1) = r(\delta_i) * f(g(\sum_j w_{ji} * a_j(t))) \tag{2.1}$$

First the refractory time for node i is taken into account, and it is calculated by inputting the remaining refractory time for the node, δ_i, into the refractory time calculation function r.

$$r(x) = \begin{cases} 1, x \leq 0 \\ 0, x > 0 \end{cases} \tag{2.2}$$

If the node has any refractory time left, its activity level will be 0 after the current time-step, and will stay at that level for subsequent time-steps until δ_i has been counted down to 0.

Next, the second factor is calculated by first multiplying the weight on each incoming link between node j and node i, with the activity of node j in the current time-step.

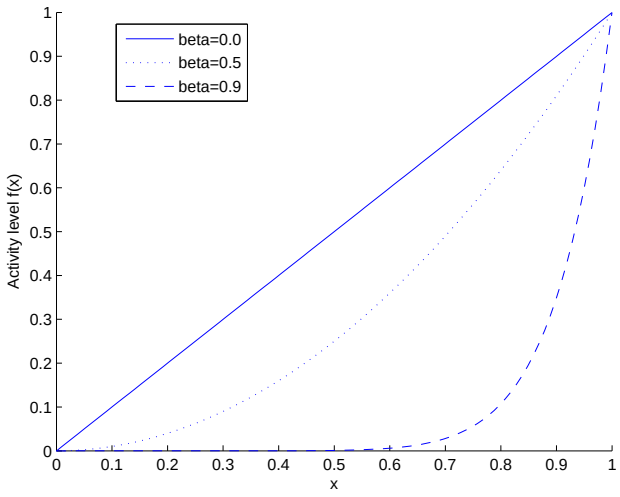

Figure 2.6: Comparison between different values of β.

This weighted sum of inputs to node i is then run through the sigmoid function g, which will produce a value in the interval (0, 1).

$$g(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

The result of the sigmoid function is in turn fed into the attention level function f. The purpose of this function is to raise the activity of a node according to the attention level parameter, β, which has a value in the range [0, 1).

$$f(x) = x^{\frac{1}{1-\beta}} \tag{2.4}$$

A value of β close to 0 will give all the nodes in the network the same amount of attention, meaning that their activity levels will hardly be changed at all with this function. A value close to 1 will put all the attention on the nodes having the highest activity level. This function will affect the activity in such a way that the more attention a node gets, the more its activity value will be raised, and vice versa. The graphs in figure 2.6 shows how the activity level is affected by different values of β.

After the activity level for node i has been calculated, the node's refractory time is updated in order to simulate that it has rested for another time-step.

$$\delta_i(t+1) = \begin{cases} \delta_i(t) - 1, \delta_i(t) > 0 \\ 0, \delta_i(t) \leq 0 \end{cases} \tag{2.5}$$

When all nodes in the network have gone through these calculations, the activity of all nodes in the network is normalised in the following manner.

$$a_i'(t+1) = a_i(t+1) * \frac{\alpha}{\sum_j a_j(t+1)} \tag{2.6}$$

This ensures that the total amount of activity in the network is preserved. This amount is determined by the value of α, which is a in the interval $[0, +\infty)$.

2.5 Node activation

After the activity of the network has been calculated, it is time to check if any nodes have activated, or fired. A node fires when its activity level reaches a threshold value, here denoted by the parameter γ. Each node is checked in turn, and if it has fired, two things happen. The first is that its activity level is set to 0 and the network is re-normalised, using equation 2.6, to account for this change in the total network activity. The second thing is that the refractory time for the node, δ_i, is set to so that $\delta_i = \delta$, where δ is a parameter of the network that decides the cool-down time for a node before it can fire again.

Keeping γ above $\alpha * 0.5$ will guarantee that there are never more than one node that fires at a time. With $\gamma < \alpha * 0.5$ there can be multiple nodes that fire at the same time-step, and it is up to the implementation to filter out the unwanted nodes. One way to do this would be to just pick one of the nodes on random, another way would be to pick the one with highest activity level.

2.6 Training

The network is trained using input from the user such that when the user rates a link from node j to node i, the weight of that link, w_{ji}, is changed in the following way.

$$\Delta w_{ji} = \eta * \mu \tag{2.7}$$

Here the network parameter η represents the rate at which the network will learn, and has a value in the range $[0, \infty)$. A higher value will yield more drastic changes in the way the network behaves, while a smaller value can be used for fine tuning. The variable μ contains the value of the input from the user. A negative rating by the user will give a negative value of μ, and the link will become weaker. In the same way, a positive rating will strengthen the link, while a neutral rating will set μ to 0 and the weight won't change at all.

At one point, it was tested if the weight of a link should only be allowed to be in a certain interval, such that if the interval is [-1.5, +1.5] and the weight gets rated to a value of -1.9 for example, it will be clamped to -1.5. This however would leave the possibility of all weights having the same value, which would render the randomised start values useless. The benefit of using clamping would be to avoid some links creating extremely high or extremely low activity values, thereby obstructing the other link values in the same calculation. After some testing, the clamping was dropped as it wouldn't be necessary if sane parameters were used. For example, having a learning rate that diminishes as time goes by would avoid the effect of link weights running wild.

Chapter 3

Implementation

After having explained the logic and functioning of the algorithms used in the previous chapter, the time has now come to describe the practical part of the thesis. In this chapter the various steps towards a fully working trainer application are shown in detail along with explanatory figures. The following text will describe shortly how the implementation of the algorithms was done, but mostly the building of the tools used to create the GUI of the trainer application.

3.1 Preliminaries

The language chosen for the implementation was Lisp, specifically the dialect Common Lisp which includes many useful functions and data structures that speeds up the process of creating an application. The reason for choosing Lisp was that the final application was originally meant to be connected to STEP[6], a natural language interface to databases that is also written in Lisp. This would provide the application with the ability to store and fetch data in a natural way, but this idea was dropped when it became clear that it would take too long to implement in addition to the other coding involved. With exception for the STEP integration, there were two goals of the implementation process. The first was to get a network working to verify the popping behaviour described in chapter 2. The second goal was to have a functional training program that should be fairly intuitive , so that a person using it should be able to train their own network with a minimal amount of knowledge of the underlying implementation details. Before coding started, a crash course in Lisp was necessary, reading tutorials and writing simple examples as well as studying other people's code. However, most of the Lisp programming was learnt during coding on the project.

3.2 How the work was done

Work on the implementation started by analysing the theoretical ideas and equations already written down, and come up with a design that would convert them into an application that was simple yet powerful. The first thing to be done was a neural network structure with accompanying functions to be able to create and modify the network as well as to run calculations on it. Creating the network structure was simply a matter keeping a list of node objects in a network object. Both the node and network types

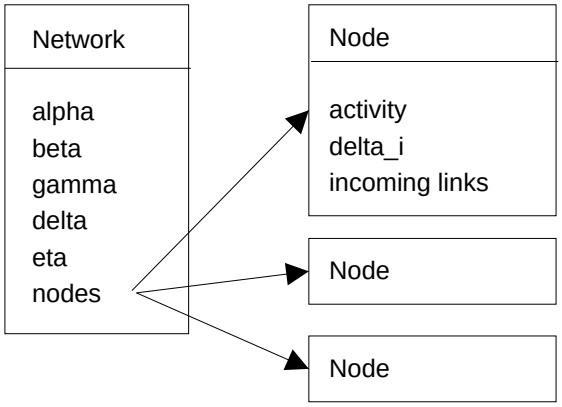

Figure 3.1: Relationship between the network and node structures.

were given variables corresponding to their theoretical counterparts. Figure 3.1 illustrates the relationship between the two types.

Having done the structural part, the next step was to make the network functional by implementing the equations described in chapter 2. This part was pretty straightforward, basically just taking the equations and putting them into functions, adjusting the variable names to suit the syntax of Lisp. The functions were kept as short and basic as possible, most of the time only calculating or changing one thing in order to keep the code readable. Testing and debugging was initially done through the interactive Common Lisp interpreter, but as more and more functions needed to be tested together it became tedious and so a simple text user interface was created. This interface worked well and helped testing and demonstration significantly. It was however difficult to get an overview of the network components, such as parameter values and node activity levels. These problems lead to the idea of having a graphical user interface, or GUI for short, instead.

The main features of the GUI would be to display the network nodes, adjust the network parameters, run calculations on the network to see which nodes fire, and to create, load and save networks. Since development was done on Linux, the GUI would have to use X calls to display its graphics. CLISP, the implementation of Common Lisp used in the development process, comes with a version of CLX, a library containing simple routines for X programming. These routines won't create widgets [1] automatically, so the CLX user will have to either use an existing toolkit or implement his own. For this project, a few different toolkits were evaluated but they were all either outdated and not working, or they were too complex and needed extra libraries to run. Therefore the decision was made to create a lightweight toolkit with only the widgets necessary for the GUI.

After creating an initial mock-up of the GUI, it was clear what different widgets needed to be created, namely the following.

- Window: a container for all other widgets.

[1] A widget is a part of the interface that the user interacts with like a button or a menu bar

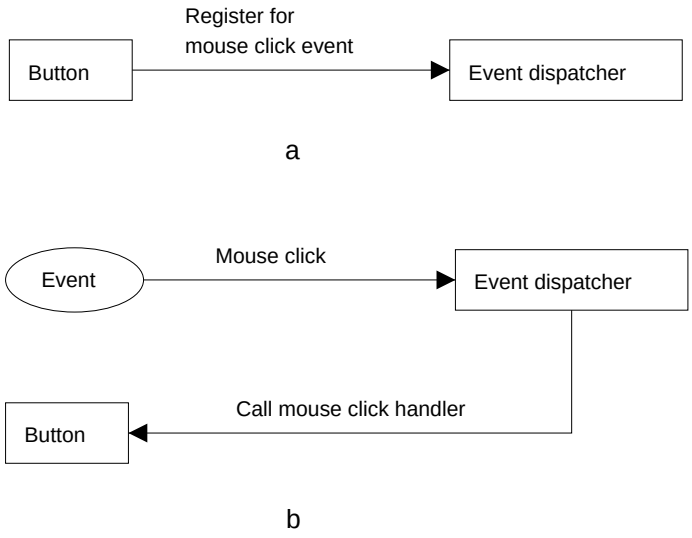

Figure 3.2: Event handling by the GUI.

- Label: a visible string of text that cannot be edited.
- Text box: a box where the user can type text, can have multiple lines.
- Button: an area which calls a function when clicked.
- Scrollbox: a text box with two buttons to easily change a numeric value.
- Network box: a special widget containing all the network nodes.
- Network node: a special widget representing a network node, shows a color corresponding to its activity level and responds to mouse clicks.

Besides the widgets, the central part of the toolkit would be the event dispatcher. Each widget is interested in certain types of events like mouse clicks or keyboard input. It registers these types with the event dispatcher, as well as telling it what function, or event handler, to call when an event occurs. The dispatcher keeps all this information in a lookup table and when for example the user clicks on a button with the mouse, that particular button's mouse click event handler is called. Figures 3.2a and 3.2b illustrate this process of registration and handling of events.

After the toolkit was considered completed, it was time to put together the GUI. By this time, a second mock-up had been created which incorporated the idea of using the GUI as a trainer for the network as well. This changed the layout slightly but without introducing any more widget types. The toolkit proved to be easy to work with and the GUI was quickly constructed. After all the widgets had been put into place, it was time to code the logic of the application, now known as the trainer.

The trainer, seen in figure 3.3 is divided into several areas. On the left side the user can set network parameters as well as change the properties of the currently selected

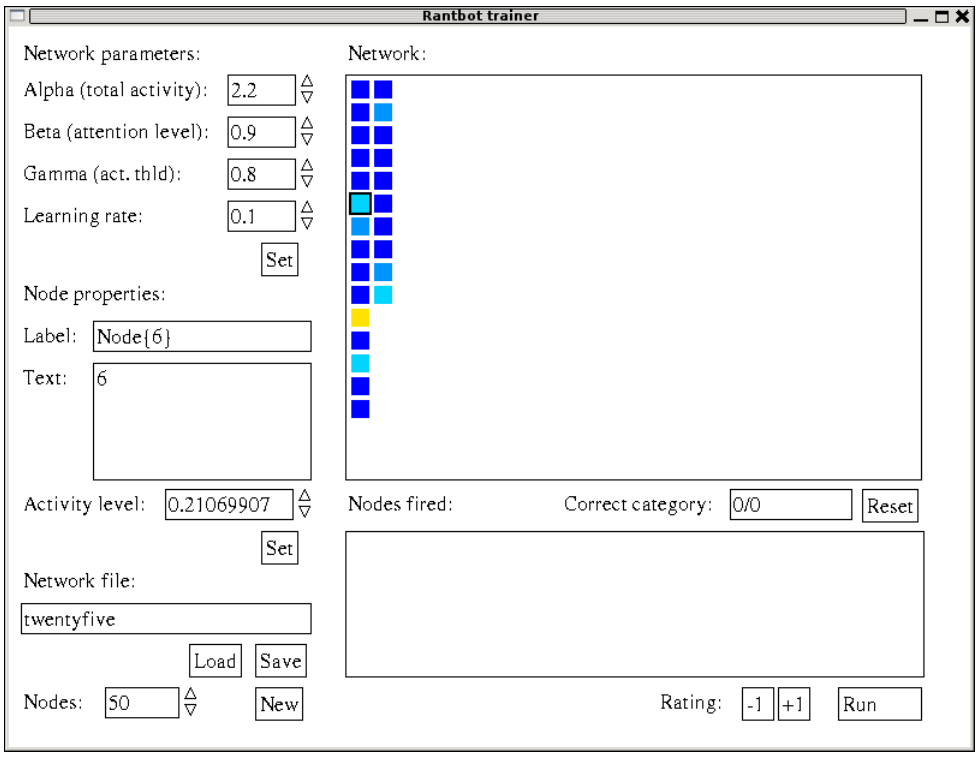

Figure 3.3: Screenshot of the trainer application.

node. In the bottom left corner are facilities for creating a new network, load an existing network from a file and also save the current network to a file. The right side of the interface consists of an upper and a lower part. The upper part is an overview of the network nodes and their activity. When the user clicks on a node, that node will be selected and its properties can be seen and edited on the left side as described earlier. In the lower part there are controls for running a round of calculations, where any nodes that fire as a result of the calculations are displayed in a text box, as well as rating the link between the next most recent and the most recent nodes that have fired.

3.3 User scenario

A typical user scenario would look something like the following. The user creates a new network by specifying the number of nodes in the bottom left corner, then clicking the "New" button. The network overview will then update to show all the nodes of the newly created networks, where each node is labelled "Node-X", where X is the number of the node. After that, the user wants to change the alpha parameter of the network and increases the value in the text box by 0.2 through using the scrollbutton. The user then presses the "Set" button below the parameters and the network is updated internally. By clicking each node and editing their properties, the user replaces all of the default labels and adds his/her own text to the nodes.

After the network has been setup, the user wants to save it in its current state, so he/she types in a file name in the box below "Network file" and clicks the "Save" button. Having a snapshot of the network safely stored away, the user moves on to run some calculations to observe the behaviour of the network. After a few clicks on the "Run" button in the bottom right corner, the big text box above the button starts to show some activity. Three lines of text have been added to the top of the box, each describing the label and text of the node that fired. The topmost node in the box is the one that fired most recently, the node below being the one fired before the topmost node. The logic is implemented such that when more than one node is above the threshold value γ, one of them is picked at random as the one that fired. When clicking on of the rating buttons, -1 or +1, the link between the top two nodes is changed an amount $\delta = \eta * (-1| + 1)$.

Chapter 4

Results

After the training GUI had been built, it was time to put it to use. During the time that the algorithms described in chapter 2 were developed and polished as well as after they were considered complete, a number of experiments were performed to test it. This chapter describes these experiments; their goals, starting conditions and results. Preliminary tests were done using two simple networks of different types, each with a different purpose. Later tests were a bit more advanced with more time spent on both performing them as well as analyzing the results.

4.1 Network 1: Three Node Cycle

This first network contains three nodes labelled A, B and C, as can be seen in figure 4.1. This network was created to demonstrate how the system generates output from its input. When a concept is input to the system, the corresponding node in the network is activated. In the example network, the activated node is B. This node has a link to node C, meaning that C will absorb activity from B in the subsequent time step. In the next time step, the activity level of C reaches the threshold value for activation, gamma. This will make node C fire and thereby C is generated as output.

Two sessions were run on this network, the first one was to show that it was possible to train the network for a cyclic behaviour using the algorithms described in chapter 2. The parameters used for the first session were α=1.0, β=0.95, γ=0.6 and η=1.0. The goal of the session was to have the network go from $A \rightarrow B \rightarrow C \rightarrow A$. The method used was click the rating button (-1 or +1) until the weight of the link was at its clamped value. After one minute, the network had been trained to the desired behaviour, without any flaws.

For the second session, the goal was to test the amount of time it would take for the network to converge to the behaviour $A \rightarrow B \rightarrow C \rightarrow A$ under normal training conditions. By this is meant that the user only clicks once per link that shows up, until he/she notices that the network is fully trained. The same parameters as before were used, and the amount of time required before the goal was reached was 4 minutes, with two of the nine links having their values clamped. One thing that was noted from this session compared to the previous one was that the weights don't have to be at their extreme values in order to get the desired behaviour.

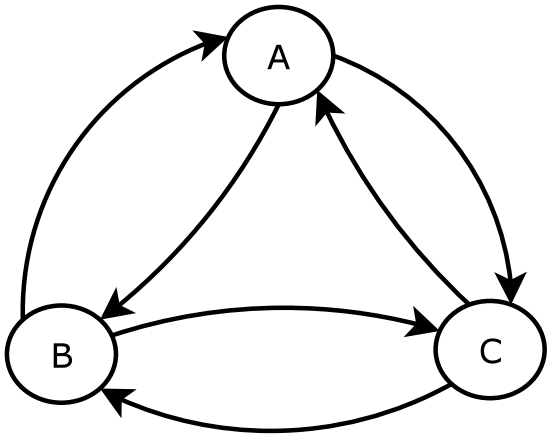

Figure 4.1: Test network with three nodes.

4.2 Network 2: Jumping Back and Forth

This network, illustrated in figure 4.2 consists of four categories with two nodes in each category. The nodes are labelled 1-8, and the categories are A, B, C and D. A consists of 1 and 2, B of 3 and 4, and so on. In the trained network, a node F2, firing after another node F1, the absolute value of the difference in node label values should equal to one ($1 \rightarrow 8$ and $8 \rightarrow 1$ are special cases). Another condition is that F1 != F2. For example, if F1 is 3 (from category B), then F2 should be either 2 (from category A) or 4. If F1 is 8 then F2 should be either 7 or 1. This type of network tests the possibility of ranting using small groups where the transitions between sentences are specified very clearly.

Nine training sessions were carried out for this network. The first session was to compare the training of it against network 1 as well as to see how fast the network would converge. After a few minutes, it was noted that the network had become stuck jumping between two nodes when the pair of links between these had become the "preferred" links with the most positive values. Session two was carried out in the same manner, but when stuck on a link it was rated negatively to get out of the loop. This didn't help however, and after a few more sessions trying different parameters and techniques, it was decided that this network would be skipped and returned to if time permitted. One final test was carried out where the weights were edited manually beforehand. This produced the wanted results, and hinted that the training process might not be suited for this type of network, at least not in the way it's been proposed and used in these experiments.

4.3 Network 3: Groups of Three

This network, which is similar to network 2, consists of three categories numbered from 1 to 3, with three nodes in each category, also numbered from 1 to 3. It is a variation on the small scale ranting, and the behaviour wanted from this network is the following:

– Any node in a category may fire after any other node in the same category.

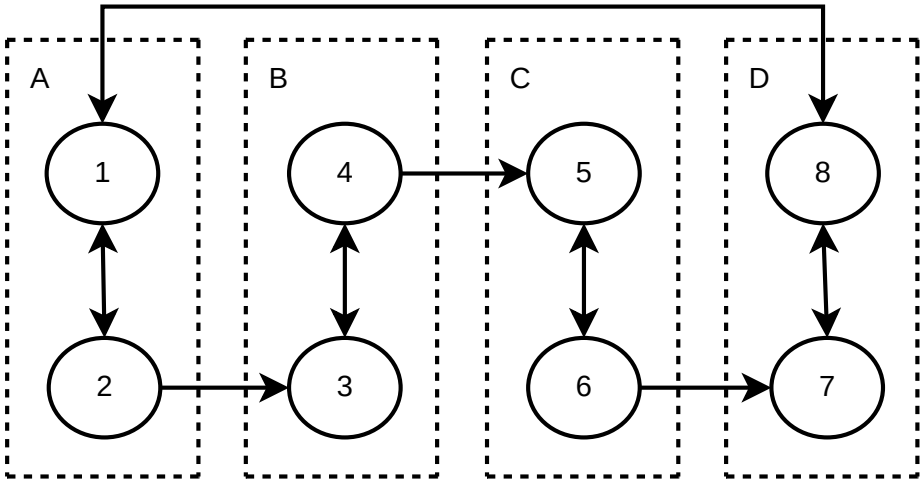

Figure 4.2: Test network with eight nodes.

- A node may not fire twice in a row.
- Node 1 in a category may not fire after a node in another category, with the exception of firing after node 3 in the previous category.
- Nodes 2 and 3 in a category may not fire after a node in another category.

The network, along with the rules stated above, are illustrated in figure 4.3. The first training session with this network yielded the same disappointing results as with network 2, namely that it got stuck. After this first session, the weights were edited by hand to check if it was at all possible to get the desired behaviour. This manual editing proved successful and the network worked as expected. The goal of the following training session was to, with the weights set to their initial values, see if it was possible to train the network to get the same results as with the manual editing. The rating was done as follows, where -2 and +2 means that the respective rating button was pressed twice instead of once.

- Going back to previous category: -2
- Wrong node leading to next category: -1
- Going to wrong node in next category via correct node: no rating
- Going to node in same category: +1
- Going to correct node in next category via correct node: +2

After 50 minutes, the results were quite close to those of the manual editing. On some occasions a node would lead back to the previous category, but that was the only fault noticed. The problem was that even though the optimal values of the weights were known, there is still the matter of training the network enough to get those values. This didn't prove easy and a person who was training a network for the first time would most likely give up pretty soon.

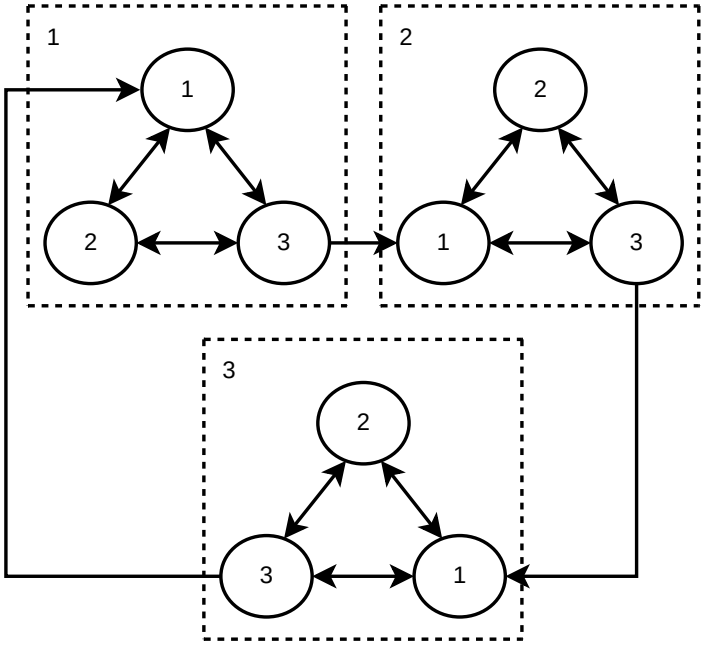

Figure 4.3: Test network with three groups of three nodes.

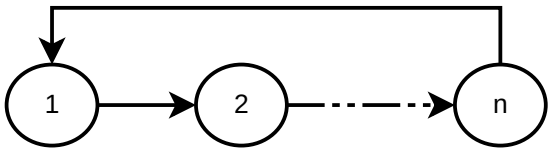

Figure 4.4: Test network with n nodes, fired in sequence.

4.4 Network 4: Sequences

This network is of the same type as network 1, but with more nodes. The goal is to have the nodes fire in a sequence, from 1 to n, then back to 1 again. See figure 4.4 for an illustration of this behaviour. These experiments will test if it is possible for the network to remember a specific sequence and being able to repeat this sequence, thus proving the priming effect mentioned in chapter 1.The training sessions on network 1 indicated that this type of network is one well suited for the training algorithm. By testing this network with increasing amount of nodes, one would get an estimate of the effect node count has on the time it takes to train the network. The method used during the training sessions was to rate -1 for correct links, -1 for incorrect links. If a correct link seemed stable, meaning that it showed up repeatedly, it wasn't rated at all in order to avoid the weights going to extreme values.

4.4.1 Five nodes

Only one training session was needed for this network. Using the starting parameters α=1.7, β=0.9, γ=0.9 and η=0.1 it took only 2 minutes until the network was fully trained to the expected behaviour.

4.4.2 Ten nodes

With twice the nodes from the previous session, the α value was brought up to 3.5 in order to keep the activity level per node about the same. For the first session of this network, γ was doubled as well to keep too many nodes from firing at once. This proved to be the wrong decision since after a while nodes stopped firing at all. A second session with the γ value lowered to 0.8 again worked better. After 20 minutes the network was fully trained. To test the effect of the learning rate, its value was doubled for the third session. After 5 minutes, the network was fully trained, thus indicating that twice the learning rate can cut the training time in half, at least for these types of networks.

4.4.3 Twenty-five nodes

The final sequential network proved to require quite a bit of training time. Starting values were α=3.9, β=0.9, γ=0.8 and η=0.1. After an hour, α was increased to 4.5 to increase the rates of fired nodes. It was increased a few more times during the session, with the final increase landing the value on 5.9. After three hours, the network was fully trained and the following things had been noted:

- A starting value of α should be chosen such that nodes fire all the time, then some more should be added to it in order to prevent choke points where no nodes fire.
- If α is increased during training, some links will need a bit of "reminding" to set them straight.
- When training a network, the learning rate should be chosen with regards to the number of nodes in the network. With a lot of nodes, each link will show up less often and therefore it will take quite a while to get its weight right.

The three networks described in this section resulted in quite different times it took to train each of them. In table 4.1, a summary of these times shows that even a small increase from five to ten nodes increases the training time by a factor of ten. Comparing the number of nodes in these two networks shows that the latter network has got four times the number of links. Granted, not all of them will be rated, but taking into account that each node only has one of its incoming links that is the correct one, the time it will take to sort our all the bad links should increase significantly as the number of nodes in the network increases.

Something else to be noted from these experiments is that there clearly is a sort of memory that is being simulated, as the sequences do not appear by chance. The training of the networks has thus resulted in providing the priming effect for each one of them.

4.5 Network 5: Sentences

For this experiment, the ultimate test, I came up with sentences that someone would be able to rant about. There are 120 sentences divided into the categories below.

Nodes	Links	Training time
5	25	2 minutes
10	100	20 minutes
25	625	180 minutes

Table 4.1: Training time for three sequential networks

- Superheroes
- World of Warcraft (a popular computer game)
- Sweden
- Movies

Within each category, there are two types of sentences; facts and opinions. From the "Sweden" category, "The capital of Sweden is Stockholm" is an example of a fact, while "Sweden is too cold" is an example of an opinion. The sentences are constructed such that for each one there is at least another sentence related to it. This is because some links should be stronger than others so that the resulting rant isn't just random sentences that are thrown out. For example, the sentence "The Exorcist is the scariest movie ever made" can be followed by "Horror movies of today are not the least bit scary". Not all of the sentences will have this clear connection, but rather be linked by a common word or context. This is illustrated by the sentence "Superman's alter ego is Clark Kent", which can be followed by "Spider-Man is the alter ego of Peter Parker". This link might not be as strong as the one between the first sentence and "Superman's costume looks a bit boring". There is also at least one sentence in each category that can be followed by a sentence in another category, so that there is a possibility that each category will be included in a rant. Since this experiment is still rather basic and the number of sentences are relatively few, any resulting attempt to a rant won't be very sophisticated. The purpose of the experiment is to show that it's possible to train and use a network for ranting, which could then be the first step towards a real rant bot.

The criteria for rating the links in the network are:

- Same category without common context: don't rate (until later on, when it seems more stable)
- Same category with common context: +1
- Different category without common context: -1
- Different category with common context: +1
- Same node as the previous one: -1

The experiment is highly subjective; deciding if a link is good or bad is up to the person performing the training. The purpose of it is to show how a typical training scenario might play out.

There were a few goals set for this experiment, namely those described below.

1. Check that at least some links get rated +1
2. See if it's feasible to train the network in 8 sessions (4 hours).
3. If not fully trained after 4 hours, see how stable the network has become in that time.

By looking at the previous experiments, having a high α and a low γ is good in the beginning, until most of the bad links have been sorted out. Likewise, a high η in the beginning will help achieve this goal. The learning rate will be decreased between each training session by 0.1 in order to avoid weight values running wild. Based on these ground rules and on values used in previous experiments, the starting parameters chosen were α=15.0, β=0.9, γ=0.5 and η=0.8.

After two 30 minute sessions, it was hard to detect any improvement in the network. It had started to get tedious to rate the links when a sentence shows up for what felt like the 100th time. Halfway through the eight sessions, there is some noticeable difference. A few chains of nodes whose links have been rated positively have started to show up repeatedly, a good sign. Also, as more and more links get rated, nodes that haven't been seen before start to pop up. After the sixth session, it has become even more noticeable that the training give results, showing more and more correct links. The last two sessions showed no big improvements besides a few more new nodes starting to appear.

What can be said from this experiment? There are a few things that were noted:

- Even though it was just half an hour at a time, it became very tedious and boring, and even painful to do more than two shifts at a time. Having the user sit and train the network until finished will require another approach except for those who are very patient.
- The interface doesn't do much to increase the training experience, it needs to be something more visually stimulating and easier to use. Quicker response from button press to finished processing would help as well.
- The parameters, besides η, could use some tweaking. Threshold could be higher, since the interface showed quite a few nodes going above the threshold value each time (4-6).

After four hours, there is some noticeable difference from the untrained network. Since there are 50*50=2500 links to rate, it is hard to say whether each link will get rated at least once. This is not necessary however in order to have the network behave the way the user wants it to. Repeating the 4 hours of training might yield results that are even impressing, although if the experiment was to be done again the amount of training hours would be increased to at least 8. The learning rate could be kept high for a longer time (about half of the 8 hours) so that the network will converge quicker to a satisfactory state.

Below is a sample output of seven sentences and the categories they belong to. This sample was taken after the network had been trained for four hours.

1. Superheroes: Batman's car is a lot cooler than Robin's motorcycle.
2. Superheroes: The Green Goblin is one of Spider-Man's greatest foes.

3. Superheroes: The Marvel universe has cooler superheroes than the DC universe.
4. Sweden: There seems to be quite a lot of Swedes playing World of Warcraft.
5. World of Warcraft: Thunder Bluff is the city of taurens.
6. Superheroes: The Joker is funnier than Batman.
7. World of Warcraft: Undercity is the city of undead.

Even though the example is quite short, it should give a feeling of the progress done after the training sessions. There is still randomness left, as can be seen with sentence four following sentence three, and with sentence six popping up between two sentences in the World of Warcraft category. However, the first three sentences are an example of how the network has been trained to stay within the same category when ranting, even though there's still finetuning needed to get sentences which are better related to each other. Finally, the step from the fourth sentence to the fifth is a good example of how some sentences can lead to another subject.

Chapter 5

Conclusions

While the ultimate goal of a working chatterbot wasn't achieved, the two most important sub goals were achieved. An application was created for building neural networks as well as training these. The tests performed in the previous chapter gave an insight to how some different networks affected the behaviour of the system. While they were mostly subjective, there are some conclusions that can be drawn from them. These tests showed that the method might not suit for all networks while it works really well for others, like the sequential networks. Other things discovered during the experiments were that in the beginning of training, it doesn't matter that much what the parameters (besides learning rate) are, since you're just going to weed out the unwanted links at first. Later on, when the network is starting to behave, it's time to fine tune the parameters until the behaviour is as close to perfection as it can get.

One type of networks tested were sequential networks, and while having a network go in a sequence might not be that useful for ranting, it could be good for example when training a robot to do something in a specific order. For example, move to point A, then to point B and so on. The most important thing to be noted from the experiments on the sequential networks was that the priming effect was achieved. This priming would be more complicated to achieve for the MDP approach, having to consider a possibly infinite amount of previous states. The method used in this thesis shows that neural networks makes the priming more natural.

One type of network that could work as a base for the final rantbot was tested as well. It showed some promising results indicating that a rantbot is certainly possible to create using the method, even though it takes time and patience to train it using supervised learning.

5.1 Limitations

One of the sub goals that weren't achieved involved connecting the system to a database and keeping the node data stored in that database, instead of on disk. Having this database would make nodes and parameters easier to edit, and it would be a better interface for retrieving the data. It would also have been interesting to see how STEP would have been integrated with the rantbot.

Creating a chatterbot that can rant, the other sub goal that wasn't met and ultimately the primary goal of this theses, would have required quite a bit more time to complete.

There would be fine-tuning of the algorithm and parameters, gathering data to have something to rant about and a lot of testing. If the bot was to be connected to an Internet chat room and respond to other user's input, that would need a lot of work as well.

Regarding testing, with only the trainer GUI, there are a lot more tests that can be made. The effect of each the various parameters could have been studied in more detail in order to find the best settings for these when training an arbitrary network. In hindsight, the implementation of the trainer application and fine tuning of the algorithms took up a bit more time than desired. Had things gone smoother regarding those two parts, more interesting and perhaps better results would most probably have been acquired.

5.2 Future work

The experiments presented in this report have been performed using a fairly small number of nodes, the maximum being 120. In a real environment the number of nodes might number in the thousands, or tens of thousands depending on the application area. This puts a lot of pressure on the scalability of the algorithm used for calculating activity levels. It should prove fairly simple to test this by first generating some networks with varying amounts of nodes in them and, since the GUI is decoupled from the rest of the network logic, create a small program which runs the application a specified number of times and measure the time it takes to run.

Since the algorithm is calculating over a number of nodes, it would be interesting to investigate the possibility of making it parallellized. For example, if a computer has two CPUs, one of them could run calculations on half of the nodes, while the other CPU deals with the other half. The potential increase in processing power would enable even larger networks to be trained and tested. The parallellization does not have to take place on the same computer however, but could be made distributed over several computers connected through the Internet, or any local network.

Another form of parallelism is related to the training of a network. In this thesis, only a single person has been training the networks at a time. Since different individuals can potentially create completely different results, having two or more people training the same network simultaneously (each person using their own GUI) could provide for a more generic network. The user interface for this approach would not necessarily have to be the same for each person, but could be accommodated for different environments. If there is a large population of trainers, some of them could even be computer controlled. This might be good in some cases where a random factor is needed for the network to work correctly.

One of the networks tested in this thesis involved having each node represent a sentence and linking together sentences by training the network. If the network was instead divided into nouns, verbs, adjectives etc. it would be interesting to see how well the method is suited for training the network to form sentences of its own. Surely it would be a time consuming task, but since there are grammatical rules for how to form correct sentences, one approach would be to create another program which would train the network. Another approach would be to use the method described above with simultaneous training by different users.

As already stated, the trainer GUI really needs some work to minimise the risk of people getting bored and frustrated while training their networks. It should be less cluttered, more intuitive and more responsive. Basically, it should be easier and more fun to use. A whole new approach to the layout and function of the GUI is probably needed, and it would definitely take some time to research, design and develop this GUI.

Chapter 1 mentions using the chatterbot in an Internet chat room with other (human) participants. This would be an interesting experiment to perform with a fully functioning rantbot. If the bot could respond to user reactions and feed it into its algorithm, one could test for how long it could go on and still make some sense. A similar and simpler experiment would be to take the output of the rantbot and post it on the website mentioned in the same chapter, and study the resulting comments that other people post.

Chapter 6

Acknowledgements

I would like to thank Mike for all the help, tips and feedback he's given me throughout the work on this thesis.

References

[1] NE Nationalencyklopedin AB. Nationalencyklopedin. http://www.ne.se/jsp/search/article.jsp?i_art_id=287002 (visited 2008-05-21).

[2] Signature 'hootie1233'. The daily rant – the uk downward spiral. http://www.thedailyrant.co.uk/rants.php?title=The%20UK%20downward%20spiral (visited 2008-04-05).

[3] M. Sandberg I. Milstein. An association-based approach to semantics. http://www-nlp.stanford.edu/courses/cs224n/2000/magnuss/report.pdf (visited 2007-03-06).

[4] R. G. Palmer J. Hertz, A. Krogh. *Introduction to the theory of neural computation*. Addison-Wesley, Redwood City, Calif., 1991.

[5] H. G. Loebner. Loebner prize for artificial intelligence. http://www.loebner.net/Prizef/loebnerprize.html (visited 2008-05-23).

[6] M. Minock. A step towards realizing codd's vision of rendezvous with the casual user. *Proceedings of the 33rd International Conference on Very Large Data Bases*, Sep 2007.

[7] P. Norvig S. Russel. *Artificial Intelligence–A Modern Approach*. Pearson Education, Inc., Upper Saddle River, N.J., 2003.

[8] R. S. Wallace. How it all started. http://www.alicebot.org/articles/wallace/start.html (visited 2008-03-02).

[9] J. Weizenbaum. Eliza–a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9:36–45, Jan 1966.