Self-Organization in Artificial Neural Networks with Biologically Inspired Spike-Rate Learning

Anders Hjellvik

July 4, 2011

Abstract

Artificial intelligence and learning is a growing field. There are many ways of making a computer program learn, in most cases one have a specific problem one wants to solve and do not really care how it is solved. This thesis have a specific problem, but the main focus is on how it is solved. One of the most exciting ways to learn is by the so called unsupervised learning methods where programs/agents learn without any human interaction. Psychologists and Neurologists have long tried to understand how the human brain works, but due to its complexity there are still some obstacles left before we will be able to simulate the different functionalities of the brain. This thesis is an attempt to get one step closer to solving the problem of how learning happens and memories form. If we were to be able to simulate human learning in a machine there is no telling where it could end. Jørn Hokland has put forward three learning rules that may describe how learning happens. These rules will be examined and then used in an artificial neural network with the intention to control a simulated robot. Artificial neural networks (ANNs) are more or less inspired by the biological neural networks (BNNs) found in humans and animals. As we will see, this thesis seeks to be one of the more biologically inspired.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will first give a short introduction to biological- and artificial neural networks. Then the background for- and the aim of this project will be described.

1.1 Neural Networks

The central nervous system consist of the brain and the spinal cord. This is a system of neurons (nerve cells) connected together into networks. What we see, hear, feel, think and our memories to name a few, are all the result of neurons getting activated in different ways in different parts of our brain. Basically who we are is determined by how our neurons are connected and how strong the connections between each neuron is. Who we are changes over time as the strength of the connections between each of our neurons is updated. We assume that these updates are based on a limited set of rules, but no one has yet found a set of rules that can describe how this learning happens. Hokland has put forth three rules that may describe this learning, and these are the rules that I am going to use later in my efforts to control a simulated robot. But first I will describe the basic principles of neural networks.

1.1.1 Biological Neural Networks (BNNs)

A neural network is composed of a huge amount of interconnected neurons. In humans each neuron receive input from 1 to about 100 000 other neurons[1]. There are different types of neurons, but in general a neuron consist of a neuron body, dendrites, and an axon, see Figure 1.1. The **dendrites** are usually the area of the cell that other neurons connect to. The dendrite's job is to receive signals from other neurons and/or the environment (light, sound etc), and then transmit these signals further to the cell body. If the input to the cell body is sufficient the **cell body** generates an electrical signal called the **action potential**. This event will often be referred to as the neuron **firing** or **spiking**. When a neuron fires the generated signal is transported along the **axon** to the axon terminals and further to other neurons. The axon of one neuron and the dendrite of another neuron are usually not physically connected and the space between them is called the synaptic cleft. The synaptic cleft together with the pre- and postsynaptic terminals makes up a **synapse**.

When there is no input to the cell, the membrane potential[2] is stable, typically between -40 and -90 mV [10][3]. This is called the *resting potential*. There are two types of input, *excitatory* input which leads to a depolarization[4] of the membrane potential (the potential becomes less negative). The other type is *inhibitory* input which leads to a hyper polarization[5] (membrane potential becomes more negative). The most interesting of these are the excitatory input, this because when the cell depolarizes and reach a membrane potential called

[1] Purves2007 page 7[10].

[2] The *membrane potential* of a cell is the difference between the internal and the external voltage across the cell membrane.

[3] Purves, Dale, Neuroscience, Fourth Edition (Sinauer Associates, Inc., 2007) p. 25

[4] More Na+ ions are entering the cell than K+ ions leaving.

[5] More K+ ions are leaving the cell than Na+ ions entering.

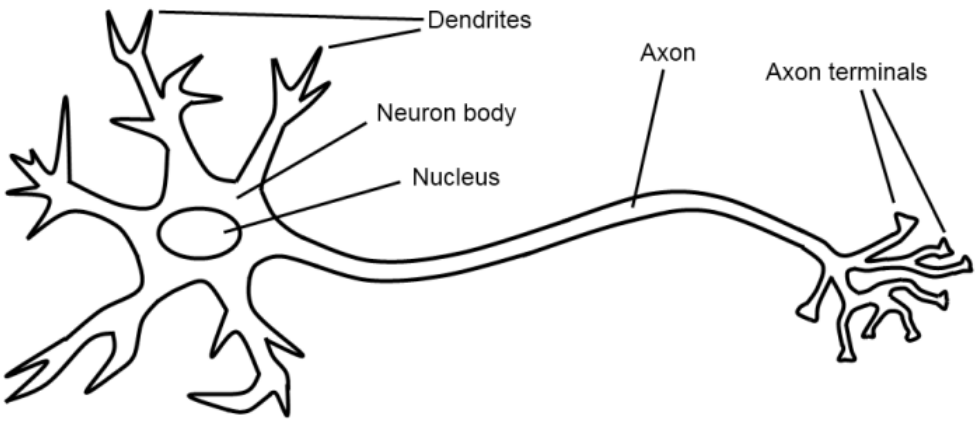

Figure 1.1: The basic parts of a neuron.

the *threshold potential* [6] it will "burst" and the membrane potential suddenly jumps to about +35 mV. After this it will rapidly decrease again and settle at the resting potential. This sudden burst is what releases the action potential. The strength of the output signal is the same whether the input is way beyond the threshold or just reaches it. The important part is the ability to transport the signal across the synaptic cleft, this is referred to as the strength- or efficacy of the synapse, in artificial neural networks it is also often referred to as the weight of the synapse.

1.1.2 Artificial Neural Networks (ANNs)

An artificial neural network is a computational model inspired by the biological neural networks. The description of the BNN above is a vastly simplified model summing up the important principles that is commonly used when designing artificial neural networks. To model an ANN we define neurons and connections between them, and then we specify how the network is going to learn, which is where Hokland's learning rules come into play. The most important part is the synapses (the connections between neurons), these are the only things that change during the simulation. Figure 1.2[7] shows the logic structure of an artificial neuron. The neuron receive inputs from n other neurons. Each of these inputs are multiplied by their respective *weights*, and accumulated to the *net input*. The *net input* is then put through an activation function. The output of the activation function is often used directly as the output of the neuron. Another more biological way of doing it is to use a threshold function that checks whether the activation of the neuron reaches a certain threshold and output a specific value if the threshold is reached and zero otherwise. Section 1.2.1 will describe the neuron model used in this thesis.

[6] Typically around -50mV
[7] The figure is taken from http://en.wikibooks.org/wiki/File:ArtificialNeuronModel_english.png

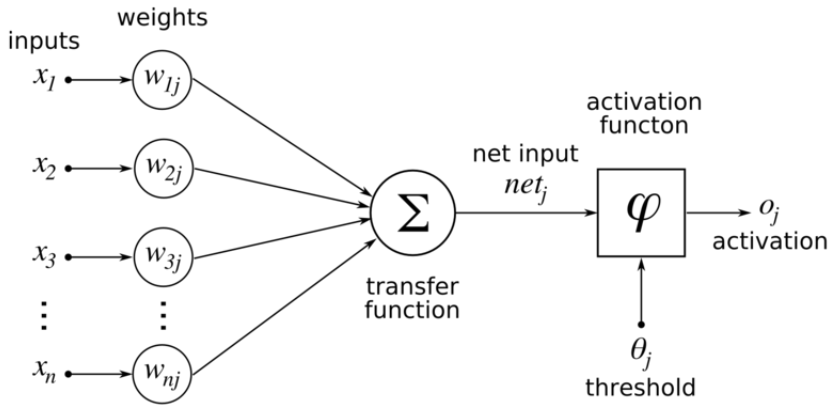

Figure 1.2: An artificial neuron.

1.2 The Background for the Project

This project is based on an earlier project where a fellow student Stian Holmås and I studied Izhikevich's spiking neuron model[5], and the properties of different neuron types. The same spiking model is used in this project and it will be described next.

1.2.1 The Spiking Neuron Model

As mentioned, a common way of determine the output of a neuron is to simply send the input through a sigmoid activation function. This can work very well in many situations but it is not the best way to simulate cortical neurons. The spiking neuron model used in my thesis is based on the spiking neuron model first proposed by Izhikevich[5], and later modified by Muresan and Savin[9]. This model is designed for the purpose of simulating cortical spiking neurons [6].

Whether a neuron is to fire or not is determined by the formula:

$$v' = 0.04v^2 + 5v + 140 - u + I \tag{1.1}$$

$$u' = a(bv - u) \tag{1.2}$$

$$\text{if } v \geq 30 \text{ mV, then } \begin{cases} v \leftarrow & c \\ u \leftarrow & u + d \end{cases} \tag{1.3}$$

where v is the membrane potential, u is the recovery variable, I is the sum of the postsynaptic currents according to Equation 1.4 below. $a,\ b,\ c$ and d are constants that determines the neuron type. We get back to these in the next paragraphs. Equations 1.1, 1.2 and 1.3 are from the Izhikevich[5] model, and calculates the neurological membrane potential in a neuron before, during and after a spike. If we compare Equation 1.3 to the description of BNNs from Section 1.1.1, it is important to note that +30mV is not the threshold but rather the amplitude of the membrane potential when the neuron is firing (spiking). The threshold is dynamic around -55mV [6], see Figure 1.3.

$$I = \sum_{i=1}^{n} psc_i \tag{1.4}$$

$$psc_i = A_{Syn} W_{Syn} g(E_{Syn} - v_{post}) \tag{1.5}$$

$$g' = -\frac{g}{\tau_{Syn}} \tag{1.6}$$

$$g = g + 1 \tag{1.7}$$

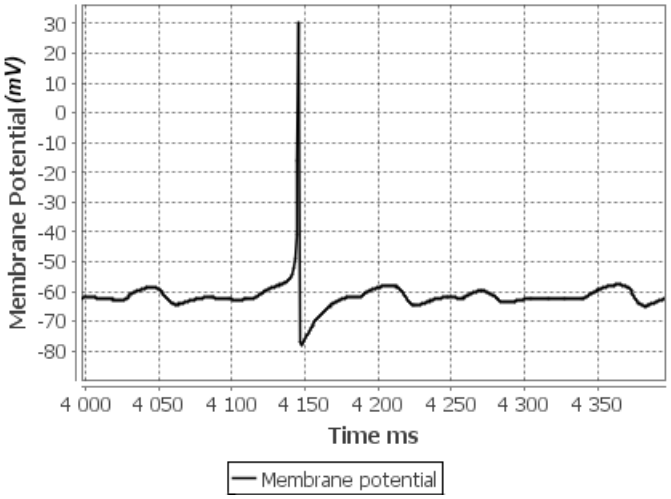

Figure 1.3: The membrane potential of a RES neuron.

Equations 1.4, 1.5, 1.6 and 1.7 are taken from the model by Muresan and Savin[9], and calculates the net input current I based on the synaptic input to the neuron.

- A_{Syn} is the "amplitude parameter"[9], and is used to tune the strength of a synapse. This may also be referred to as the *coupling strength*. It can be defined independently for each kind of synapse i.e. synapses between different layers can have different A_{Syn} values, or synapses from RES neurons to RS neurons can have one A_{Syn} value, while RES-RES synapses have another. RS and RES neurons are described in the next paragraphs.

- W_{Syn} is the *synaptic weight* also referred to as the *synaptic strength* or *efficacy*. This is one of the most important variables. It is this variable that will be manipulated by Hokland's learning rules which in turn will determine if the ANN can self-organize. The initial weight can be set randomly or one can set it to a default value.

- g models the synaptic conductance (which depends on the concentration of neurotransmitters[8] in the synaptic cleft) in each synapse. Equation 1.6 models the conductance decay (reabsorption of neurotransmitters), while 1.7 models the conductance increase caused by a presynaptic spike (release of neurotransmitters), which is set to a fixed value of 1 per spike.

- E_{Syn} is defined as the *reversal potential* of each synapse[9]. It is set to 0 for excitatory synapses, and -70 for inhibitory synapses.

- v_{post} is the membrane potential of the postsynaptic neuron.

- τ_{Syn} is the rate of synaptic conductance decay (neurotransmitter reabsorption rate). Higher values of τ_{Syn} will prolong the effect of each spike and increase synaptic strength in a way similar but not equivalent to increasing the A_{Syn} value.

By setting the parameters a, b, c and d we can generate many different neuron types, in my thesis I have only uses two different types which will be described next.

[8]Neurotransmitters are the chemicals that transmit the action potential signal from one neuron to another[10].

Regular-Spiking (RS) Neurons: The Regular-Spiking neuron is based on models by Izhikevich[5], slightly modified by Muresan and Savin[9]with $c = -70$ instead of $c = -65$. RS neurons demands a pretty high input to get started, but when the input is high enough they spike much more frequently than the RES neurons described below. So one can say that the RS neurons are more unstable, and they have a tendency to become epileptic, or saturated[9]. One of the problems of RS neurons is that they are not self sustained, i.e. they are not able to keep each other going over time, they will either saturate or loose all their activity. They need additional input either from other neuron types, the environment or some background current. The RS neurons are defined by the following values:

- $a = 0.02$
- $b = 0.1$
- $c = -70$
- $d = 8$

Resonator (RES) Neurons: The resonator neurons are more sensitive to low inputs but less responsive to large inputs. The benefits of this is that a network of RES neurons is self sustained, i.e. after the initial activation the RES neurons manage to keep each other firing at a healthy frequency. The second benefit is that RES neurons will seldom saturate, they manage to keep the activity within bounds even with large input currents. The drawback is that we often would like the neurons to be more responsive to changes in the input current.

The RES neuron was first described by Izhikevich[5] under the name *RZ neuron*. I use the same version as Muresan and Savin[9], with $c = -70$. The RES neuron is defined by the following parameter set:

- $a = 0.01$
- $b = 0.26$
- $c = -70$
- $d = 2$

1.3 The Problem Description

There are two main goals of this project. The first is to create a biologically inspired artificial neural network using Izhikevich's[5] spiking neuron model [10], and Hokland's synaptic learning rules [4]. This network is first going to be used to examine Hokland's learning rules[11] to verify that the rules work as intended in practice.

The next step is to use the network as a controller for a simulated robot[12], often referred to as the agent. The agent got two arms, each controlled by four muscles which in turn are controlled by the ANN. The aim is for the network to self-organize so that the agent learns a movement pattern that brings it closer to a goal position. In its simplest form the agent must learn to push up & forward followed by down & backward, where the amount of force and the timing are crucial. The driving force of the agent is a wish to get to a food source while trying to minimize the energy consumption (applied force). This will be described in more detail in Section 2.1.2.

[9] Often defined as having a spike rate above 300 Hz [9].
[10] Adopted by Muresan and Savin[9]
[11] The learning rules are described in Chapter 3
[12] The robot and its environment are described in Section 2.2

1.3.1 Unsupervised Learning

The task of making a simulated robot move forward may not seem very difficult, but it depends on how complicated the movement pattern to be learned is and how much information is given to the robot. My task is not mainly to get the agent to move forward, but to make it learn by the same mechanisms that humans learn. We usually distinguish between three forms of machine learning[11]:

- **Supervised learning** is based on a supervised training phase. One typically knows what output any given input is supposed to produce, and one will update the strength of the synapses (weights) based on how close to the wanted output the result was.
- **Reinforcement learning** is learning without a teacher available, but one will receive rewards for desired behavior and adjust the weights accordingly.
- **Unsupervised learning** resembles reinforcement learning, but the agent have to develop a sense of what is good and what is not before it can recognize a reward.

My agent learn in an unsupervised way, i.e. it is completely self taught. It will not get any information on whether its actions are good or bad. The way the learning is going to happen is that the agent will gradually form a memory of events by using Hokland's learning rules to update the synaptic strengths, and then after some time it will in theory be able to predict what actions will lead to the wanted need reductions which is the ultimate goal. The learning rules are described in Chapter 3. In particular the Skinner rule is designed to minimize genetically programed needs, see Section 3.2.

Chapter 2

The Application and the Mechanical Model

This chapter describes the application I have created and the external tools used for 3D simulation and other visual aspects. It also describes my agent, the muscle model used to simulate its muscles, and the needs that drives the agent to act.

2.1 The ANN

The ANN consists of a set of classes written in Java. I chose Java as the programming language because it is the language I am most familiar with and there are a lot of easy available libraries that will also be of great help.

The neural network part of the system consist of a GUI where the topology of the network is chosen and the most important parameters are set. The topology is defined in a .txt file using keywords for layers and synapses (an example of such a file can be found in Appendix A.2.1). When the parameters in the GUI is set as wanted, one can hit the *"Run Network!"* button to start the simulation. When the simulation starts, a new SpikingNetwork instance is generated which in turn uses a NetworkParser instance to parse the topology file and generate Layers, Neurons and Synapses. After this is done the SpikingNetwork instance start looping a procedure which update the input to the nodes, the spike rate of each neuron and the weight of each synapse (Appendix A.2.2 shows the source code for the main loop) . For a closer look at the GUI and a short explanation of all its parameters and functions, see Appendix A.1. Figure 2.1 shows a class diagram of the ANN.

2.1.1 Network Topology

The network topology (structure) obviously has a great effect on the behavior of the network. When not using evolutionary algorithms I have to try to make wise decisions based on my intuition and previous research. The topology I started with was based on Figure 2 from Hokland2011[4] which in turn is based on cortical research. The topology simulates the six layers of the neocortex. Based on what we know about the connections between these layers, Hokland has put forth a hypothesis on which of his learning rules applies in and between each layer. I will try a few different topologies based on this, as we will get back to in Chapter 5.

2.1.2 Needs

The needs of my agent is what motivate it to act, and they are an essential part of the learning process. Needs are coded as high activity (spike rate) in the neurons. In the biological realm high activity means high energy consumption and that is not desirable. The agent will try to save energy and therefor minimize the different needs (the Skinner rule as we will get back to later).

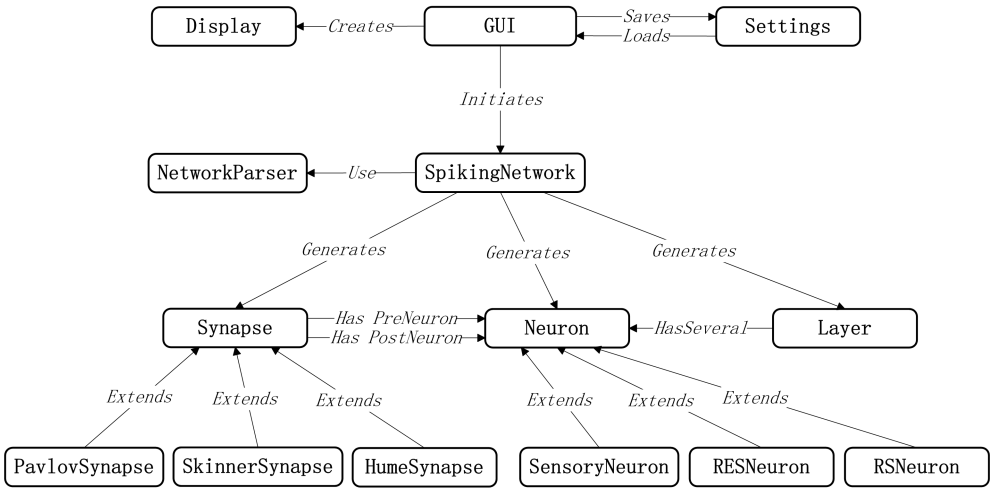

Figure 2.1: Class diagram of the ANN.

My agent have two major needs, the first one is to minimize the use of muscle force, and the second is to get to a food source. It is clear that these two needs are in conflict, i.e. the best way to minimize the force need is to do nothing, so if the agent did not have the other need it would soon die of starvation. The challenge is to balance the needs as to minimize the total energy spent while still getting to the food source.

The distance need: The agent knows its own coordinates, and it knows the coordinates of the food source, the greater the distance the stronger the need. This need is divided into two different distance needs. The first is to get forward, the second is to get up from the ground. The challenge is to find a good way to represent this distance so that there is a significant drop whenever the agent moves a little closer to the goal. If just using the raw distance data the need would decrease too slowly to learn anything from it. So I needed to amplify the need reductions. What I did was to send the distance through a function so that the need decrease fast in the beginning and then gradually slower as the agent approaches the goal. This way most of the learning would occur in the beginning and later it would be more like fine tuning the weights. I used the following formula for this purpose:

$$D_{Distance,t} = a \cdot g_t^b + c \cdot g_t + d \tag{2.1}$$

where $D_{Distance,t}$ is the distance need at time t, g_t is the normalized[1] distance from the agent to the food source at time *t*. *a, b, c* and *d* are constants (not to be confused with the a, b, c and d parameters of the neuron model). Note that when using a high value for b it could lead to very high values of $D_{Distance,t}$ if the agent moves backwards so that $g_t > 1$, this could lead to the neuron constantly firing. Figure 2.2 shows an example of a distance need function. The same kind of function is also used for the other needs and for the length and muscle velocity inputs from the agent to the ANN.

Minimizing muscle force: The point of this need is to make the simulation more realistic, for example people and animals that have a very limited food supply will try not to waste any energy. In practice in my simulation this need to minimize the use of muscle force will hopefully lead to smooth movements of the arms. If my agent would learn how to move forward and there were no force reduction need, it is likely that the agent would use a much bigger force than necessary for every movement.

[1]The current distance divided by the start distance

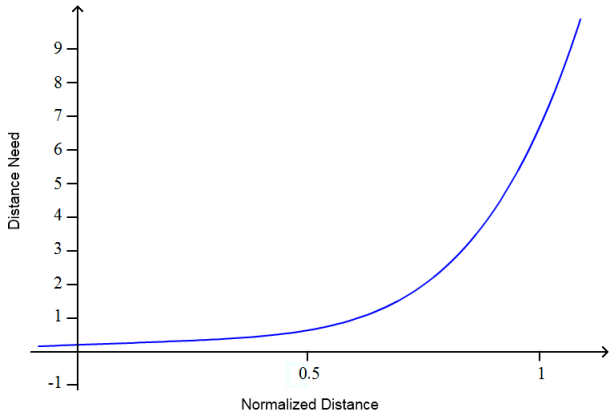

Figure 2.2: The distance-need function, in this case a=5, b=5, c=0.5, d=0.2

2.2 The Mechanical Model

For the mechanical simulation I have used Cyberbotics' Webots v6.3.3.[2]. This software is build on the Open Dynamics physics Engine[3], and simulates an environment with gravity and other natural forces making it very realistic. The main reasons for using Webots was that it is fairly easy to use for modeling robots in a realistic environment, and it is also easy to integrate Webots with the ANN. When using Webots it is important to make sure that the Webots clock is synchronized with the ANN clock.

2.2.1 My Agent

I have chosen to create a very simple robot (agent) using the Webots application. The agent basically consist of a rectangular body, two arms and a GPS sensor, see Figure 2.3. The arm joints are rotational servos with two degrees of freedom so that the arms can be moved up, down, forward and backward. There are a total of four motors controlling eight muscles (each motor controls one muscle of the left and one of the right arm, in other words the arms are synchronized). If the experiment is successful for this setup the plan is to see if the agent is able to synchronize the arms on its own. This is a much more complicated task, and it would not be wise to start there for several reasons. It would be like trying to teach a child to run before it could walk. It would also introduce a whole lot of new things that could go wrong, and it would make it really hard to study and tune the neural network.

Creating the models in Webots is not very hard, but when using forces to control the servos many things can go wrong. Initially I had some problems getting the agent moving forward at all because of the amount of force needed to lift the front of the robot of the ground and then push back. This would demand a really high spike rate for the down neurons and at the same time low spike rate at the up neuron. I could of course increase the maximum force available, but that would lead to unstable behavior and the arms would often dislocate due to to large forces in certain situations. The problem of not moving forward is that the spike rate in the distance need neurons will never decrease, and we will never get any learning in those synapses. So if I can not get some initial movement the learning rules will not be able to do much good. The solution to this problem was to remove some of the friction by adding some passive wheels to the agent. That way I could keep the forces under control and the agent would be able to move under normal circumstances.

[2]Cyberbotics Webots: http://www.cyberbotics.com/
[3]Open Dynamics Engine: http://www.ode.org/

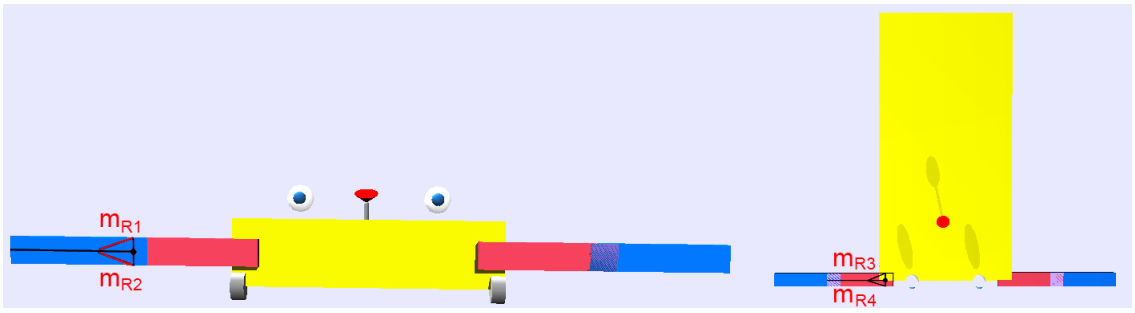

Figure 2.3: My agent with imagined muscles drawn on. On the left a front view, and on the right a top view. $m_{R1}$and m_{R2} controls the up and down movement- while m_{R3} and m_{R4} controls the backward and forward movement of the right arm.

2.3 The Muscle Model

To make the simulation as close to reality as possible I have used a biologically inspired muscle model to simulate the muscle behavior. The muscle model is adapted from Axelsen [1]. The model is originally based on Kandel's [8] description of biological muscles. What the model does is to transform the spike rate of the motor neurons in my ANN, the muscle length and the muscle velocity into muscle forces.

Kandel describes three different muscle fibers that most mammalian muscles are composed of[4]. The different fibers mainly differs in how much neuron activation they need to contract, and how powerful the contraction is. Axelsens' model is intended to simulate slow-twitch muscle fibers (Type I fibers)[5], but the other two fibers, the so called fast-twitch fibers (Type II fibers)[6], can also be simulated as I will get back to later in this section.

The muscle model consists of three active forces and two passive forces based on the spike rate of the associated neuron, the muscle length and the muscle velocity. So these three variables have to be calculated and normalized before they are used to calculate the different forces.

Spike rate, muscle length and muscle velocity:

The spike rate is obtained from the ANN and I will get back to how it is calculated in Chapter 4.

The muscles used is purely logical structures. Initially a max and min angle is set for each muscle, and the muscles attachment point is defined. Based on the attachment points and the current angle of the arm, the length of the muscle can be calculated according to the following equation:

$$L = r * 2 * \sin((\frac{\pi}{2} - m_{Direction} * P_{Arm})/2)) \tag{2.2}$$

where L is the muscle length. P_{Arm} is the current position (angle) of the arm measured in radians according to Figure 2.4, the position is obtained from the Webots application and it is independent of the muscles, which is the reason for the $m_{Direction}$ variable which is the muscle direction, the value is either 1 or -1 depending on which muscle we are looking at, i.e. it is 1 for muscle m_{R1}, and -1 for the opposite muscle m_{R2} (Figure2.3).The distance to the attachment point r is also described in Figure 2.4. In order to normalize the length I have used the following formula:

$$L_{Norm} = (L - L_{Min})/(L_{Max} - L_{Min}) \tag{2.3}$$

[4]Kan00[8], page 684.

[5]Slow-twitching fibers change slowly in response to the spike rate of the motor neurons, they can sustain relatively small forces over a long period of time.[8]

[6]Fast-twitching fibers change rapidly in response to the spike rate of the motor neurons, they can generate relatively large forces over a short period of time.[8]

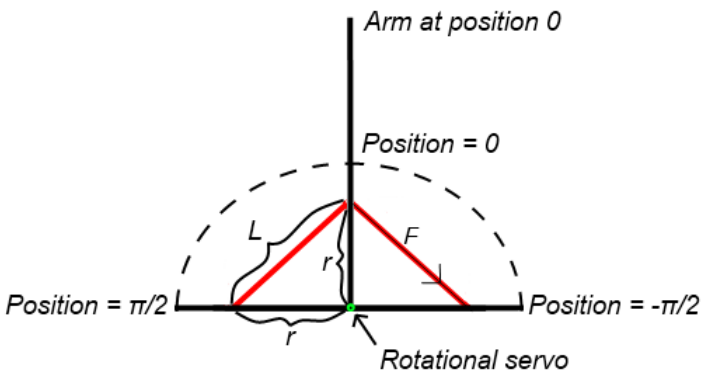

Figure 2.4: Logic structure of the muscles, L is the muscle length, r is the length form the servo to the muscle attachment point, the position is the angle of the arm in radians, the start position of the arm is shown in the figure. F is the force applied to the muscle.

where L_{Max} and L_{Min} are the maximum- and minimum muscle lengths which can be calculated by replacing P_{Arm} of Equation 2.2 with the predefined max and min angle respectively.

The muscle velocity is defined in the model to be relative to 0.5. That is, values greater than 0.5 represent a positive velocity, and values below 0.5 represent a negative velocity. It is calculated according to the following formula:

$$V_{Norm} = 0.5 + ((\frac{\triangle L}{\triangle t}/(L_{Max} - L_{Min}))/2 \tag{2.4}$$

where V is the normalized muscle velocity, $\triangle t$ is the time frame.

Calculating the Active and Passive forces

Now that we have the three variables which the muscle model force calculation is dependent upon, we can calculate the three active forces and the two passive forces. The following equations are taken from Axelsen2007[1]. They are developed by Axelsen and two experts in the field of human movement science, Gertjan Ettema and Beatrix Vereijken.

The first active force is basically the normalized spike rate (S_{Norm}) of the motor neurons (the input from the ANN to the muscle):

$$f_{a1}(S_{Norm}) = S_{Norm} \tag{2.5}$$

The second active force is dependent upon the normalized muscle length (L_{Norm}):

$$f_{a2}(L_{Norm}) = -(1.4L_{Norm} - 0.9)^2 + 1 \tag{2.6}$$

The third active force is dependent on the normalized muscle velocity (V_{Norm}):

$$f_{a3}(V_{Norm}) = \begin{cases} (\exp(V_{Norm} - 0.572957))^7, & if \quad V_{Norm} < 0.5 \\ -10(V_{Norm} - 0.7)^2 + 1, & if \quad 0.5 \leq V_{Norm} < 0.7 \\ 1, & if \quad V_{Norm} \geq 0.7 \end{cases} \tag{2.7}$$

The first passive force depend upon the muscle length (L_{Norm}):

$$f_{p2}(L_{Norm}) = (L_{Norm} - 0.25)^6 \tag{2.8}$$

The last passive force depend upon the muscle velocity (V_{Norm}):

$$f_{p3} = 0.5V_{Norm} + 0.25 \tag{2.9}$$

When all these are calculated we can calculate the current contribution to the force by the following formula:

$$F' = f(S_{Norm}, L_{Norm}, V_{Norm}) = (f_{a1} \cdot f_{a2} \cdot f_{a3} + f_{p2} \cdot f_{p3}) \cdot F_{Max} \tag{2.10}$$

where F_{Max} is the user defined maximum force. The current force does not only depend upon the current situation, it also relies on the previous forces. This is taken into account in the following formula:

$$F_t = (1 - \alpha_m)F_{t-1} + \alpha_m F'_t \tag{2.11}$$

where F_t is the total force generated at time t. $\alpha_m \in (0,\ 1)$ is a constant that controls how much to weight the current contribution to the force.

As mentioned earlier I have edited this formula a little to simulate both types of muscle fibers. The way I have set up my system is that I use two motor neurons for each muscle, one to simulate the slow-twitching fibers and one to simulate the fast-twitching fibers. To simulate the fast-twitching fibers the presynaptic weights of the fast-twitching neurons are set higher than the weights of the slow-twitching neurons. In addition to this the α_m variable from Equation 2.11 is increased for the fast-twitching muscles to emphasize the current contribution to the force more. To calculate the total force produced by both muscle fibers the spike rate of each motor neuron is fed to Equation 2.10 to generate F'_1 and F'_2. These are then used in Equation 2.11 with different α_m values to obtain $F_1$and F_2 as shown below:

$$F_{Tot} = F_1 + F_2 = ((1 - \alpha_{m1}) \cdot F_1 + \alpha_{m1} \cdot F'_1) + ((1 - \alpha_{m2}) \cdot F_2 + \alpha_{m2} \cdot F'_2) \tag{2.12}$$

where F_{Tot} is the sum of the forces generated by both the types of muscle fibers.

Since my mechanical model is driven by rotational servos the final step will be to transform the calculated force into torque. That is done according to the following formula[7].

$$\tau = F \cdot r \cdot \cos((\frac{\pi}{2} + m_{Direction} \cdot P_{Arm})/2) \tag{2.13}$$

where τ is the torque, recall Figure 2.4.

2.4 The Entire System

Figure 2.5 shows a class diagram of the entire system, at the top we got the Webots application which initiates the muscle force controller which in turn initiates the GUI class of the ANN.

[7] An alternative way of writing this formula: $\tau = r \cdot F \cdot \sin((\frac{\pi}{2} - m_{Direction} \cdot P_{Arm})/2)$

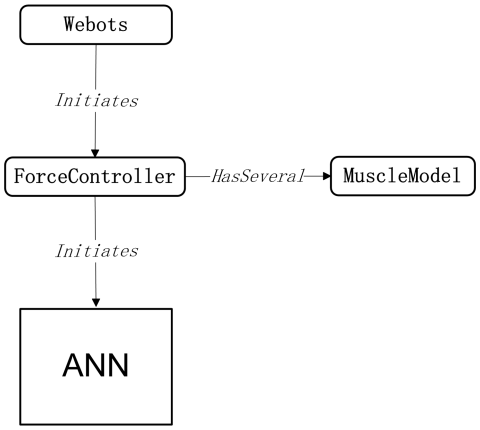

Figure 2.5: Class diagram of the application. The mechanical part of the application consist of the Webots application, and two classes I have created to control the agent. The ANN is displayed in detail in Figure 2.1.

2.5 Other Tools

I have used some external libraries for plotting different data, and generating the GUI.

2.5.1 JFreeChart

All of my graphs are drawn using JFreeChart[8]. This tool provides a lot of very helpful features, it supports a huge amount of different charts and is easy to use. In my case one of the most important features has been the zoom function, when the graphs get really big (i.e. stretches over a long period of time) it is very helpful to be able to extract small windows and look at them. Figure 2.6 shows an example of the graphs used in this report.

2.5.2 GUI Builder

Designing a GUI in SWING can be quite cumbersome, especially in my case where I am constantly adding and removing parameters and features, so I chose to use Abeille Forms Designer[9] to do the hard work. This application is a WYSIWYG[10] application and it enables me to select SWING components and place them on a canvas. The information is then stored in an .xml file which is read and parsed to objects in my application so that i can easily fetch the different components. Figure 2.7 shows a screen shot of how my GUI looks in the Forms Designer application.

[8] JFreeChart: http://www.jfree.org/
[9] Can be found at: http://www.softpedia.com/get/Programming/Other-Programming-Files/Abeille-Forms-Designer.shtml
[10] Short for: What you see is what you get.

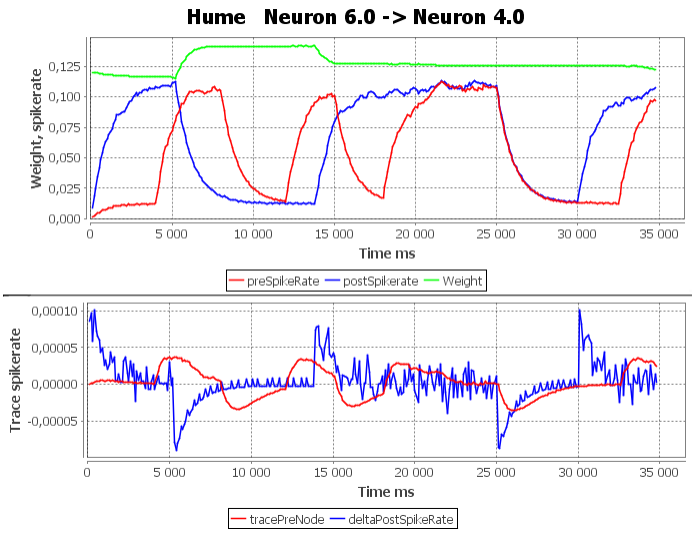

Figure 2.6: An example of the graphs drawn by the JFreeChart library. The top graph shows the spike rate of two neurons that are connected by a Hume synapse, and the weight of the synapse. The bottom graph shows the trace of the change in the presynaptic neuron, and the change in the postsynaptic neuron. The presynaptic neuron will always be displayed in red color, the postsynaptic neurons will be drawn in blue, and the synaptic weight will be displayed in green.

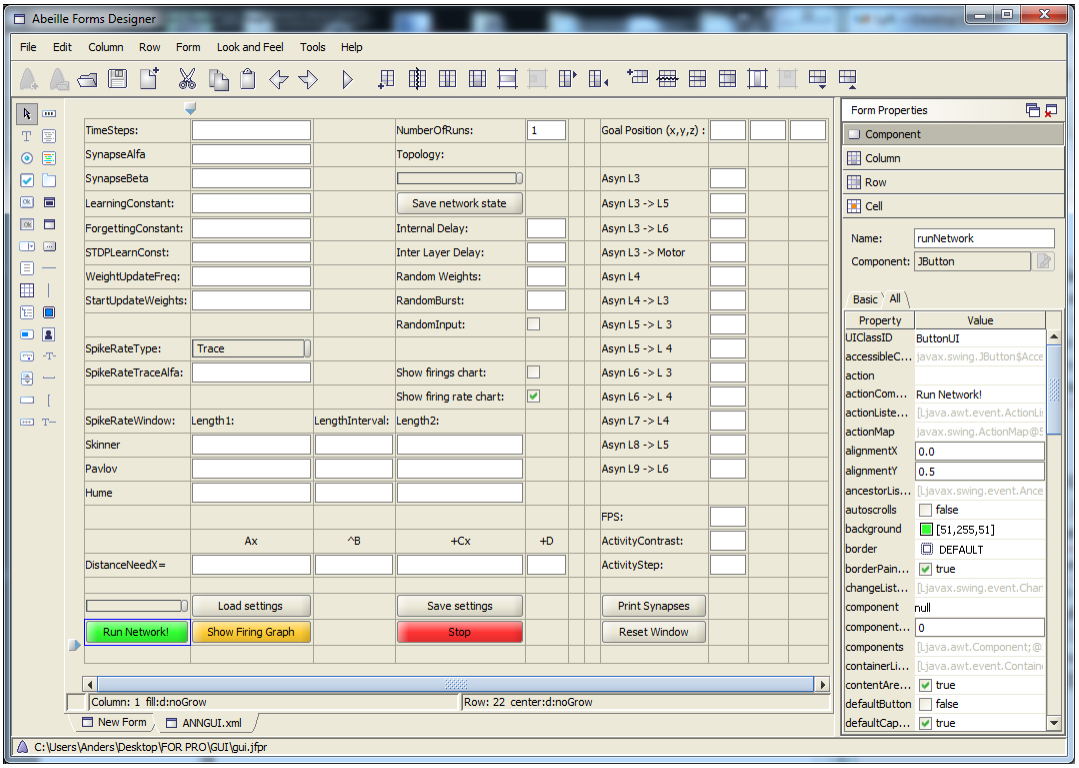

Figure 2.7: Abeille Forms Designer.

Chapter 3

The Learning Rules

The synaptic learning rules are one of the major keys in understanding how the brain works. If we knew by which learning rules the synapses of the brain learn it would be a huge step into an unknown world of possibilities. We could make computers learn in similar ways as humans and there is no telling where that could end.

This chapter describes Hokland's three learning rules and the intuition behind them. Then some challenges in implementing the rules are presented.

3.1 Motivation for the New Rules

Hebbian learning is a very popular learning method. The idea of Hebbian learning is that the strength of the synapse between two neurons is increased if they both fire at the same time[1], and decreased if this is not the case. One of the problems with this is that it is unstable because the synaptic strength often will reach infinity. There are modifications of the rule that deals with this problem but Hokland takes a different approach inspired by some of Einsteins teachings.

Hokland has proposed three alternative rules to explain how learning happens in a biological neural network. Basically, all we remember are events (state changes), and we remembered them as coming before or after other events. We learn based on our memory, and since our memory is just a series of events ordered in time, we only learn from changes[4]. This is contrary to the Hebb rule which is trying to learn within each state, rater than learning based on the changes from one state to another. Asking what led to that neuron being fired, rater than assuming since these neurons fired at the same time they must be related to each other.

In all three rules it is important that there is some time between the spike rate changes in the pre- and postsynaptic neurons. This is because we do not want to learn if the presynaptic neuron directly is causing the postsynaptic neuron to spike. It is obvious that directly connected neurons will effect each other but if we were to update the weight at such times we would soon have a problem of weights approaching infinity (like the Hebb rule). What we want to do with these rules is to update the weights when there is a distinct spike rate change in one neuron, and later another neuron is influenced by other neurons and change as well.

The next three sections will describe Hokland's three learning rules. Each of the rules uses some of the following equations to update the synaptic weights[2].

$$\triangle X_t = \left(\frac{X_t - X_{t-u}}{u} \right)^{\beta_{Syn}} \tag{3.1}$$

$$\triangle Y_t = \left(\frac{Y_t - Y_{t-u}}{u} \right)^{\beta_{Syn}} \tag{3.2}$$

[1] If delays between neurons are used this is taken into account

[2] Modified from Hokland2011[4]. In the original equations u is replaced by 1, but updating the weight every time step would not work in practice, see Section A.1 in the Appendix for an example.

$$\triangle x_t = \alpha_{Syn} \cdot \triangle x_{t-u} + (1 - \alpha_{Syn}) \cdot \triangle X_t \tag{3.3}$$

$$\triangle y_t = \alpha_{Syn} \cdot \triangle y_{t-u} + (1 - \alpha_{Syn}) \cdot \triangle Y_t \tag{3.4}$$

where X_t is the spike rate of the *presynaptic neuron* at time t, u is the update frequency [3]. Y_t is the spike rate of the *postsynaptic neuron* at time t. β_{Syn} is an odd integer used to amplify the differences in spike rate change. $\triangle x_t$ is the trace of $\triangle X_t$, and $\triangle y_t$ is the trace of $\triangle Y_t$. α_{syn} is a real number in the range $0 < \alpha_{syn} < 1$. In practice this value determines the length and amplitude of the trace, i.e. a high α_{syn} value leads to a long trace with a low amplitude, and a low α_{syn} value leads to a short trace with a high amplitude.

3.2 The Skinner Rule

The Skinner rule is the most important of the three rules. It is what drives the network to learn.

> « The basic assumption leading to the new and simple Skinner Rule is that in general all aversive stimuli are coded by high drive levels, while all forms of pleasant stimuli are coded by the absence of high drive levels.» [4]

Basically we want to use as little energy as possible, and since firing a neuron is not free it is designed to fire when there is some kind of discomfort. The rule concerns needs or wishes. The idea is that we all are born with some basic needs, i.e. the need for food. When our body is low on blood sugar our brain lets us know that we are hungry, we need to eat. The theory is that this is done by increasing the spike rate of the relevant need neurons. While the satisfaction we feel after we have eaten is due to a drop in the spike rate. So eating leads to a drop in the need neuron's spike rate, which is beneficial and the weight is updated to enforce a similar behavior next time we feel hungry. The Skinner rule is designed to only learn from satisfaction, i.e. when the spike rate goes down.

The Skinner Rule:

- **Case 1:** Postsynaptic spike rate increase followed by a later presynaptic spike rate decrease causes the synaptic strength to be increased.
- **Case 2:** Postsynaptic spike rate decrease followed by a later presynaptic spike rate decrease causes the synaptic strength to be reduced.

This can be transformed into the following equation:

$$\triangle W_{Skinner,t} = -min(\triangle X_t, 0) \cdot \triangle y_t \cdot C_{Hok} \tag{3.5}$$

where C_{Hok} is the learning constant used for Hokland's rules, for the rest of the expressions recall Equations 3.1-3.4 above. See Figure 3.1 for a graphic visualization of the Skinner rule.

The extended Skinner rule: In one of my meetings with Hokland he presented a new part to the equation. The new part is a part that draws the weight a little bit towards zero every time the spike rate of the postsynaptic neuron changes without any later event in the presynaptic neuron. If this happens a lot it is a good indicator that it is not the postsynaptic neuron that is causing the presynaptic neuron to decrease its spike rate, therefore we want to adjust the weight a little towards zero (decrease if the weight is above zero, and increase if the weight is below zero). It is very important that this adjustment is only a small fraction of the increase (decrease) of Case 1 (Case 2). In my application I control this by the "forgetting constant", C_{Forget}, Section 3.7 will show how to set this constant. The extended Skinner rule can be stated like this:

[3] In my case this value is set to 100, meaning that the weights are updated once every hundred time steps.
[4] Originally from Hokland1997[2], cited in Principia Psychologica 2010[3] page 245.

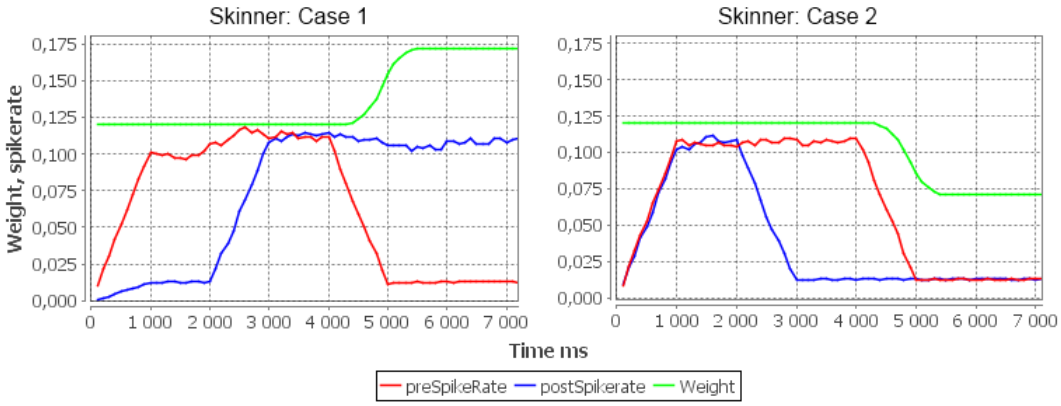

Figure 3.1: The Skinner rule.

- **Case 3:** Postsynaptic spike rate increase or decrease without any later change in the presynaptic spike rate causes the synaptic strength to be pulled slightly towards zero.

If we take this third case into account the Skinner equation becomes:

$$\triangle W_{Skinner,t} = (-min(\triangle X_t, 0) \cdot \triangle y_t - abs(\triangle y_t) \cdot W_t \cdot C_{Forget}) \cdot C_{Hok} \tag{3.6}$$

where W_t is the current weight of the synapse.

3.3 The Pavlov Rule

The Pavlov rule is intended to induce anticipation. If something often happens after something else, one might start to anticipate that if the first thing happens then the second is likely to happen soon too. The Pavlov rule is designed to amplify the cause and effect relationships between two neurons. i.e. if there is a increased spike rate at the presynaptic neuron and later an increase in the postsynaptic neuron spike rate, the weight is increased so that in the future the presynaptic neuron will have an increased effect on the postsynaptic neuron. It is only spike rate increases at the presynaptic neuron that leads to learning by the Pavlov rule.

The Pavlov Rule:

- **Case 1:** Presynaptic spike rate increase followed by a later postsynaptic spike rate increase causes the synaptic strength to be increased.
- **Case 2:** Presynaptic spike rate increase followed by a later postsynaptic spike rate decrease causes the synaptic strength to be reduced.

This can be transformed into the following equation:

$$\triangle W_{Pavlov,t} = max(\triangle x_t, 0) \cdot \triangle Y_t \cdot C_{Hok} \tag{3.7}$$

where C_{Hok} is the learning constant used for Hokland's rules, for the rest of the expressions see Equations 3.1-3.4 above. See Figure 3.2 for a graphic visualization of the Pavlov rule.

The extended Pavlov rule: In a Pavlov synapse we want to draw the weight a little towards zero if the presynaptic spike rate increases without anything happening in the postsynaptic neuron. In other words if the presynaptic spike rate seems to have little or no effect on the postsynaptic spike rate, the weight is pulled a little towards zero.

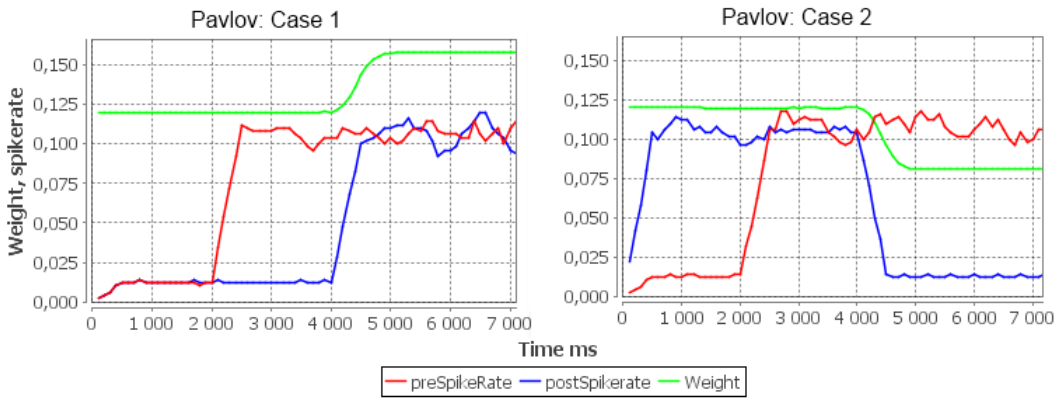

Figure 3.2: The Pavlov rule

- **Case 3:** Presynaptic spike rate increase, without any later change in the postsynaptic spike rate causes the synaptic strength to be pulled slightly towards zero.

The formula for the extended Pavlov rule becomes:

$$\triangle W_{Pavlov,t} = max(\triangle x_t, 0) \cdot (\triangle Y_t - W_t \cdot C_{Forget}) \cdot C_{Hok} \tag{3.8}$$

where W_t is the current weight of the synapse and C_{Forget} is the "forgetting constant".

3.4 The Hume Rule

The Hume rule is intended to learn concepts and form plans. To grasp the beginning and the end of something. Figure 3.3 shows an example of the intuition of the Hume rule. In my case the intention of the Hume rule is to

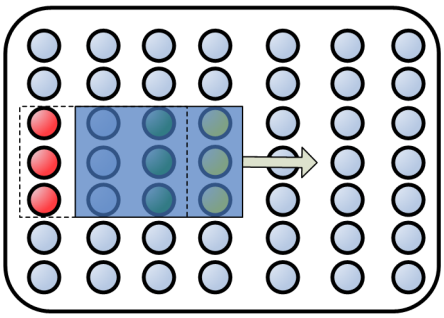

Figure 3.3: Hume example, visual field. The figure shows 7x7 neurons constituting an imagined visual field. In the figure a blue rectangle is moved to the right increasing the spike rate of the neurons it covers. Last time step (the dotted line) the green neurons increased their spike rate as a result of this. At the current time the red neurons has just been passed by the rectangle and as a result decreased their spike rate. The Hume rule would in this case increase the weights of all the synapses from the green neurons to the red neurons, while it would decrease the weight of the synapses from the green to the yellow neurons (which just increased their spike rate).

form paths of neurons activating or inhibiting each other, for example the neurons that control the up motor in my mechanical model tend to inhibit the neurons that controls the other motors.

The Hume Rule:

- **Case 1:** Presynaptic spike rate increase followed by a later postsynaptic spike rate increase causes the synaptic strength to be reduced.
- **Case 2:** Presynaptic spike rate increase followed by a later postsynaptic spike rate decrease causes the synaptic strength to be increased.

This can be transformed into the following equation:

$$\triangle W_{Hume,t} = -max(\triangle x_t, 0) \cdot \triangle Y_t \cdot C_{Hok} \tag{3.9}$$

note that the only difference from the Pavlov rule is the sign at the beginning of the equation. See Figure 3.4 for a graphic visualization of the Hume rule.

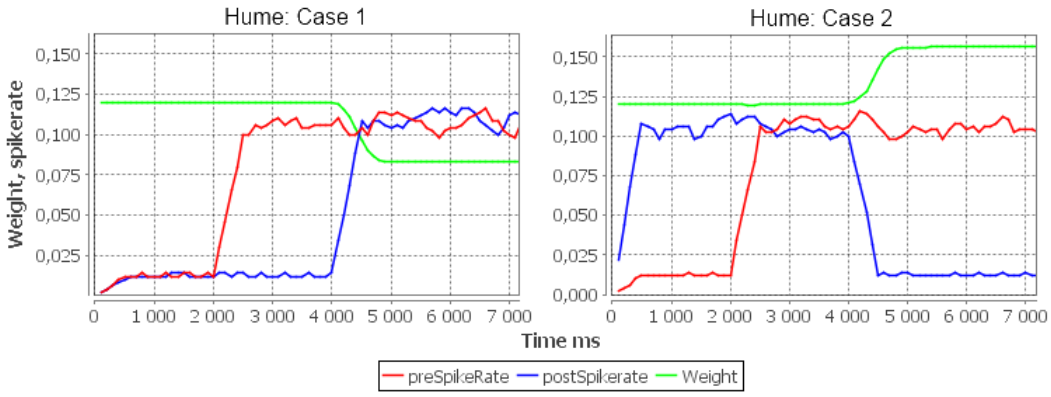

Figure 3.4: The Hume rule.

The extended Hume rule: If the presynaptic neuron increase its spike rate and the postsynaptic spike rate remains stable we assume that the postsynaptic neuron is not part of the plan/an edge, and therefore move the weight a little towards zero.

- **Case 3:** Presynaptic spike rate increase, without any later change in the postsynaptic spike rate causes the synaptic strength to be pulled slightly towards zero.

The formula for the extended Hume rule becomes:

$$\triangle W_{Hume,t} = -max(\triangle x_t, 0) \cdot (\triangle Y_t + W_t \cdot C_{Forget}) \cdot C_{Hok} \tag{3.10}$$

where W_t is the current weight of the synapse and C_{Forget} is the "forgetting constant".

3.5 The STDP Rule

In addition to Hokland's learning rules I have implemented the STDP[5] rule as described in Izhikevich06[7]. The reason for implementing this rule is that the STDP rule may be a mechanism for short term memory[4].

The STDP rule will increase the synaptic strength between two neurons if the presynaptic neuron spikes (fires) and then the postsynaptic neuron spikes within a given time bound τ. If the postsynaptic neuron spikes before the presynaptic neuron the weight is decreased. The the magnitude of the weight change is determined by the interval between the latest presynaptic- and postsynaptic spikes, according to Equation 3.11 :

[5] Spike-timing-dependent plasticity

$$\triangle W_{STDP,t} = \begin{cases} C_{STDP} \cdot A_+ \cdot e^{-s/\tau_+} & if \quad t \geq 0 \\ -C_{STDP} \cdot A_- \cdot e^{s/\tau_-} & if \quad t < 0 \end{cases} \tag{3.11}$$

where s is the time interval from the last presynaptic spike to the last postsynaptic spike, $s = s_{post} - s_{pre}$. A_+ , A_-, τ_+ and τ_- are constants used to simulate the behavior of biological synapses[6]. C_{STDP} is the learning constant used for the STDP rule. This parameter can be set to zero if one only wants to use Hokland's rules.

The total change to the synaptic strength becomes:

$$\triangle W_t = \triangle W_{Hok,t} + \triangle W_{STDP,t} \tag{3.12}$$

where $W_{Hok,t}$ is either the $\triangle W_{Skinner,t}$, $\triangle W_{Pavlov,t}$ or $\triangle W_{Hume,t}$, see Equations 3.5 - 3.10.

3.6 Challenges With Implementing the Learning Rules

This section describes some of the challenges I encountered when implementing and experimenting with the learning rules.

3.6.1 Estimating the Spike Rate

The first challenge is to find a good estimate of the spike rate, this is essential to get the learning rules to work as intended. The main problem is to get a good estimate for each time step without getting a spike rate that fluctuates to much. This problem is dealt with in Chapter 4 so I wont go into any more details at here.

3.6.2 The Skinner Weight Problem

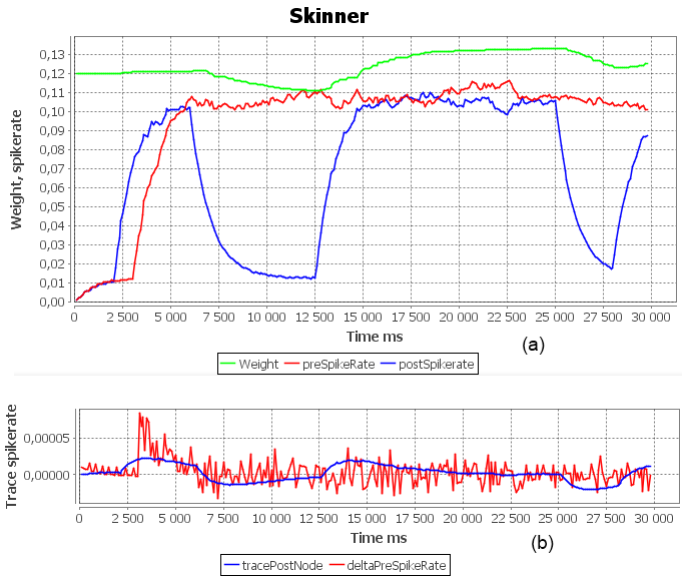

Figure 3.5: A challenge with the Skinner rule. The weight varies according to the trace of the postsynaptic neuron based on small drops in the presynaptic spike rate. Every time the red graph of the bottom figure is below zero the weight will be increased if the bottom blue graph is positive and decreased if it is negative.

[6]The constants are set according to Izhikevich2006[7]: $A_+ = 0.10$, $A_- = 0.12$ and $\tau_+ = \tau_- = 20ms$

Another challenge which I discovered during testing is that the weight of the Skinner synapses tend to follow the activity of the spike rate of the postsynaptic neuron even though there is no events at the presynaptic neuron (see Figure 3.5(a)). Recall that a Skinner synapse is set to learn only when the spike rate of the presynaptic neuron drops. The problem is that the spike rate increases and drops a little all the time. If we look at Figure 3.5(b) the synapse learns every time the change in the presynaptic spike rate (bottom red graph) is below zero, and this happens a lot. So why is this a problem only for the Skinner synapse? That is because the Pavlov and Hume synapses updates only as long as the trace is positive, and the trace is not fluctuating like the current change in spike rate is, and when the trace is positive the variations in delta spike rate cancels each other out. This problem strongly relates to how the spike rate is calculated.

A Proposed Solution to the Skinner Weight Problem

One way of dealing with this problem is to increase the β_{Syn} value of Equations 3.1 and 3.2. This would diminish the smallest changes in spike rate. Figure 3.6 shows a comparison of different β_{Syn} values. From the figure we can see that increasing the β_{Syn} value efficiently removes the unwanted weight changes, but high β_{Syn} values demands extremely high learning constants, and this makes the network pretty unstable. Very small changes in spike rate may have a huge impact. So in a network where one typically does not know how big spike rate changes one might get or one knows that the changes vary quite a bit it is almost impossible to choose a good learning constant for $\beta_{Syn} > 3$. Another problem is that the "noise" often drops as fast as the longer drops, so that if we want to eliminate the noise we will often eliminate the event too, or at least parts of it as we can see from Figure 3.6(c) where the weight stops updating while the spike rate is still dropping pretty fast. The higher β_{Syn} the less noise but the more unstable it gets, and with a learning constant at 7E56 we would expect some instability. In practice I would never recommend to increase β_{Syn} above 3.

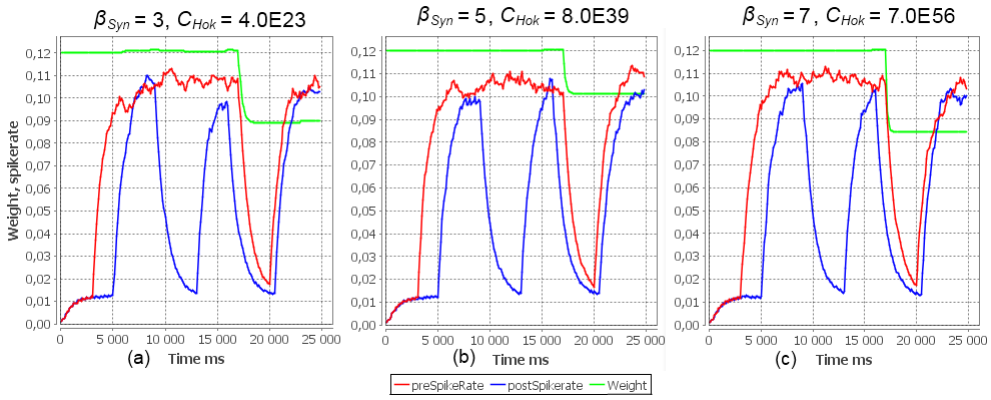

Figure 3.6: Varying β_{Syn}.

Another way to deal with this problem is obviously to try to get the spike rate to be more stable. When using trace based spike rate estimate, this can be done by increasing the α_{Trace} , see Equation 4.1. But as we will see in Section 4.3 that has its problems too, as a sudden change in spike rate would be seen as a slowly increasing spike rate over a long time window, e.g. if the spike rate increase from 10Hz to 30Hz in an instance it could take many seconds before the calculated spike rate would reach 30 Hz.

The best way to deal with this issue is to calculate the change in spike rate in a different way and no longer use the trace function of Equations 3.3 and 3.4, this is described in Section 4.4.

3.6.3 The Order of the Events Problem

When the spike rates of the presynaptic- and postsynaptic neuron change at the same time we get a large effect. Even if (in the case of Skinner synapses) the spike rate of the presynaptic neuron changes before the postsynaptic

neuron, we may get a big effect. The reason for this unwanted behavior can be seen in Figure 3.7 bottom graph, where the weight change is defined by Equation 3.5. The figure shows a case where the presynaptic spike rate drops 500 ms before the the postsynaptic spike rate change. From the trace (bottom blue graph) and the change in the presynaptic spike rate (bottom red graph) we can see that in this case the event at the presynaptic neuron may happen at least two seconds earlier and still draw the weight up quite a bit. This problem could be reduced a little by decreasing α_{Syn} of Equation 3.3, but a better way to do it is to use the spike rate estimates given in Section 4.4.

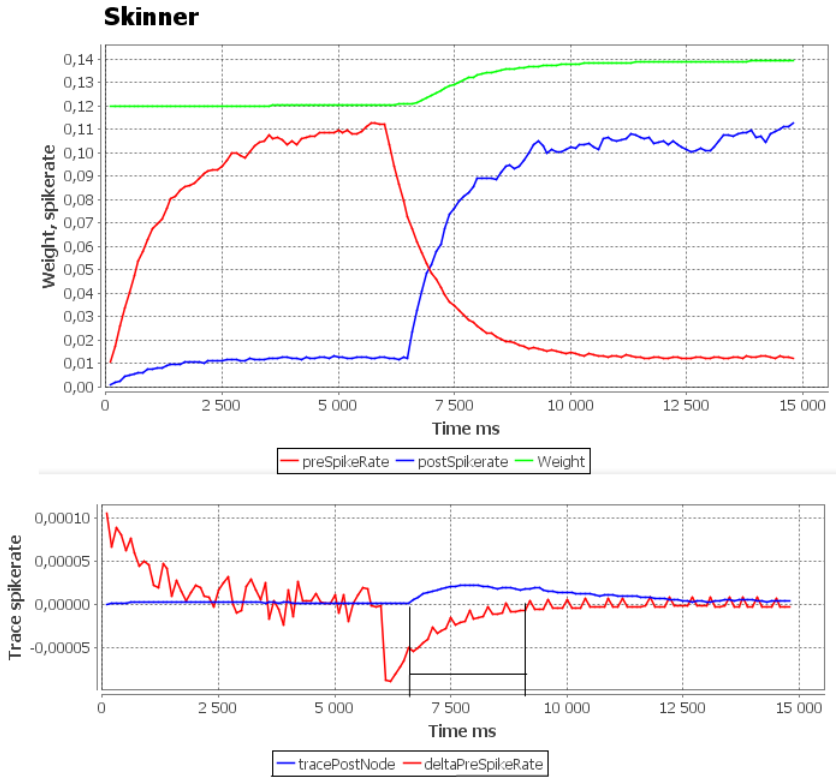

Figure 3.7: When the order of the events are wrong, or too close. The figure shows a Skinner synapse that learns even though the presynaptic spike rate change 500 ms before the postsynaptic spike rate change. The black lines on the bottom graph are drawn on to illustrate the additional time distance the presynaptic spike rate change have to appear before the postsynaptic change to avoid this unwanted weight update.

3.7 Tuning the "forgetting constant", C_{Forget}

The "forgetting constant" is the constant introduced in Equation 3.6 with the intention to control how fast the weight is to move towards zero in certain cases. The example shown here is done on a Pavlov synapse, an example of the Skinner and Hume rules can be found in Appendix A.3.

If we look at Figure 3.8, the interesting time frame is from ca 13 000 ms where the presynaptic spike rate increases while the postsynaptic spike rate remains stable. From Figure 3.8(a) we can see that a forgetting constant equal to zero results in a stable weight. We have small fluctuations since the spike rate is constantly going a little bit up and down, in the long run these changes cancel each other out in Pavlov and Hume synapses (recall Section 3.6.2 for more detail). In Figure 3.8(b) there are two things to notice when we compare it with Figure 3.8(a). First, in Figure3.8 (a) the weight starts to increase around 3000 ms, while in Figure3.8(b) the weight first decrease until the spike rate of the postsynaptic neuron also increase a lot. This shows that the initial

small increase in the postsynaptic neuron is outweighed by the forgetting constant. We also see that a single event where the synapse is to be moved towards zero cancels the initial increase entirely. These two are clear indicators that the constant is way to large. Second, the initial weight increase is reduced from about 0.04 in Figure3.8(a) to about 0.02 in Figure 3.8(b). This is due to the fact that the forgetting part is active every time the presynaptic neuron increases its spike rate. The result of this is that normal weight increases will be reduced and weight reductions will be amplified (for positive weights). This is not a desired effect, but as we can see from Figure3.8(c) it will be minimized with a lower forgetting constant. Figure3.8(c) shows the desired effect[7], here we can see that it would take about ten "forgetting events" without any other learning to cancel out the initial weight increase.

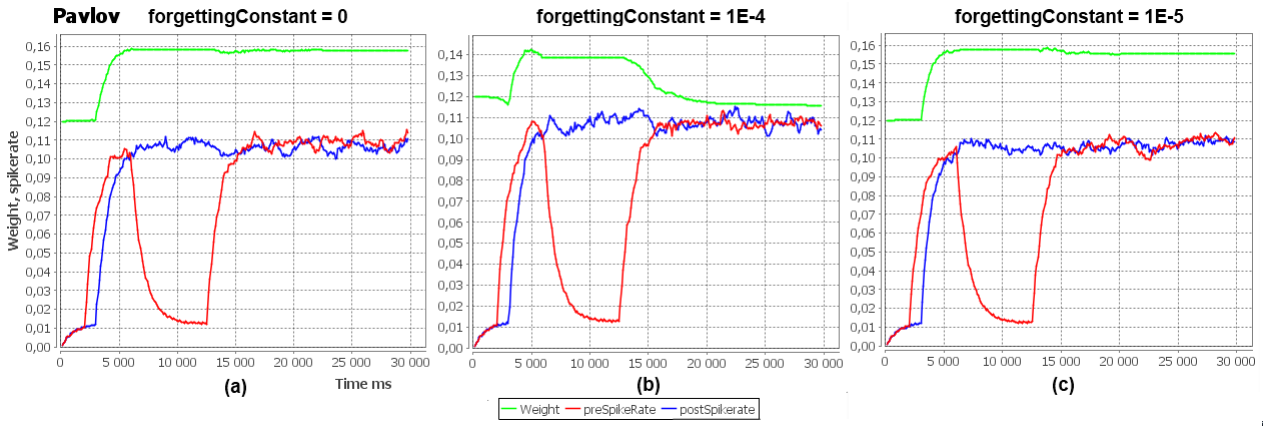

Figure 3.8: Tuning of the forgetting constant.

[7] In some cases we would like a even lower forgetting constant.

Chapter 4

From Single Spikes to Spike Rate

Hokland's rules are designed to increase/decrease the synaptic strengths based on a change in spike rate rather than single spikes. So one of the big questions then is how should the spike rate be calculated. This chapter describes a few alternative ways to do it, some better than others.

I had a great focus on run-time in the beginning due to working with much bigger networks in the project leading up to this one, but since the networks I use now is very small in comparison the run-time is not a big problem[1]. I also have to synchronize the ANN with Webots and the Webots application will not allow me to run my application faster than real time[2].

4.1 Sigmoid Weighted Window

The first way I tried to estimate the spike rate was to use a weighted history window of 64 bits. This would soon appear to be a far to narrow window size since there typically would only be a few or none spikes in each interval. The reason for using such a small window was that the spike rate is calculated for every neuron every time step (ms) and to keep a really long history would slow things down. In addition to this the size of Java Long is 64 bits and I could then use the Long as a binary queue by bit shifting one place to the left and adding a 1 or a 0 if there was a spike or not. This was done for rapid update of the spike history and spike rate calculation. But in the end the window of 64 bits was to small.

4.2 Distance Based Spike Rate

After a meeting with Hokland where we talked about other ways to calculate the spike rate so that it would not return to zero every time there was no spike the past 64 time steps. At the same time we did not want to keep a history of several seconds. One suggestion was to use the distance between the last two spikes and set the inverse of that as the spike rate. To avoid that the spike rate was constant between each spike I used the following formula:

$$SpikeRate = \frac{1}{lastSpikeDistance + max(currentTime - lastSpikeTime - lastSpikeDistance, 0)}$$

The effect of this is that if there is X time steps between two spikes the spike rate is set to 1/X and it remain the same in X time steps (unless a new spike occurs), after X time steps it starts decreasing and for each time step T after the first X time steps, the value becomes 1/(X+T), until the next spike occurs see Figure 4.1. If we compare the scale on the Y-axis of Figure 4.1 with Figure 4.2 in the next section we can see that the distance

[1] Although I noticed the speed reduction for the largest topology described in Section 5.5.

[2] The Webots EDU version that is, in the PRO version it is possible to speed up the simulation.

measure rises and falls about ten times faster than the trace based estimate. On a small time frame as shown in the figures this does not look to bad but when running this on the real network over a much longer time it will be very unstable in that it will give a to local estimate of the spike rate.

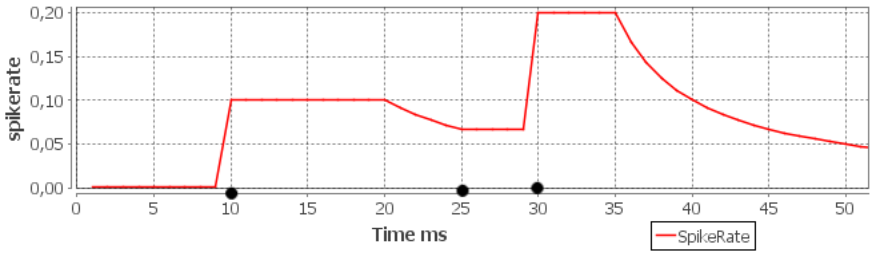

Figure 4.1: Distance based Spike Rate estimate. This type of spike rate is unstable, i.e. it may jump from zero to one in two ms if the spike rate is zero followed by to spikes in a row.

4.3 Trace Based Spike Rate

I had to find a more stable estimate of the spike rate in order for the learning rules to work as intended. This was done by calculating the spike rate using a trace function:

$$S_{Trace,t} = \alpha_{Trace} \cdot S_{Trace,t-1} + (1 - \alpha_{Trace}) \cdot \Lambda_t \tag{4.1}$$

where α_{Trace} is a constant in the range (0, 1). Λ_t is 1 if the neuron spiked at time t, and 0 otherwise. Figure 4.2 shows how each spike affects the spike rate, and how the spike rate decays after a spike.

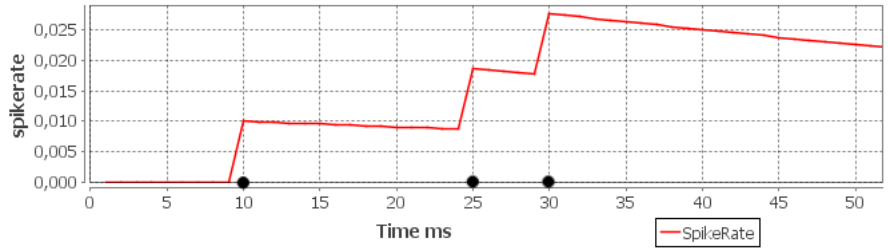

Figure 4.2: Trace based spike rate on the single spike level. The black dots after 10, 25 and 30 ms mark the time when the neuron spiked, in this case α_{Trace}= 0.99 so new spikes adds 0.01 to the spike rate.

What is good about this method is that it is very easy to calculate and can give fairly good estimates of the spike rate given the right α_{Trace} values. Figure 4.3 shows the effect of varying α_{Trace}. As we can see this way of calculating the spike rate demands a very high α_{Trace} value . Figure 4.3(a) shows a good estimate, even though it takes some seconds for the spike rate to change. Figure 4.3(b) shows a very smooth spike rate which is good for the learning rules when updating the weights, but as we can see it has gone thirty seconds since the increase and it has still not flattened out. In practice a spike rate that moves this slowly towards its stable state (about 0.11) will usually not get near it before it changes again. This way the spike rate will almost always increase or decrease very slowly. In practice this means (for Pavlov) that if there has been a presynaptic spike rate increase the weight will update based on the activity in the postsynaptic neuron until there is a decrease in the presynaptic neuron so there will always be an active trace and the weight is adjusted according to the postsynaptic neuron. We also have to lower the α_{Syn} value of Equation 3.3 quite a bit to be able to detect fast changes, and the main problem of the order of the events is even more clearly present. So all in all using such a high value is not recommended.

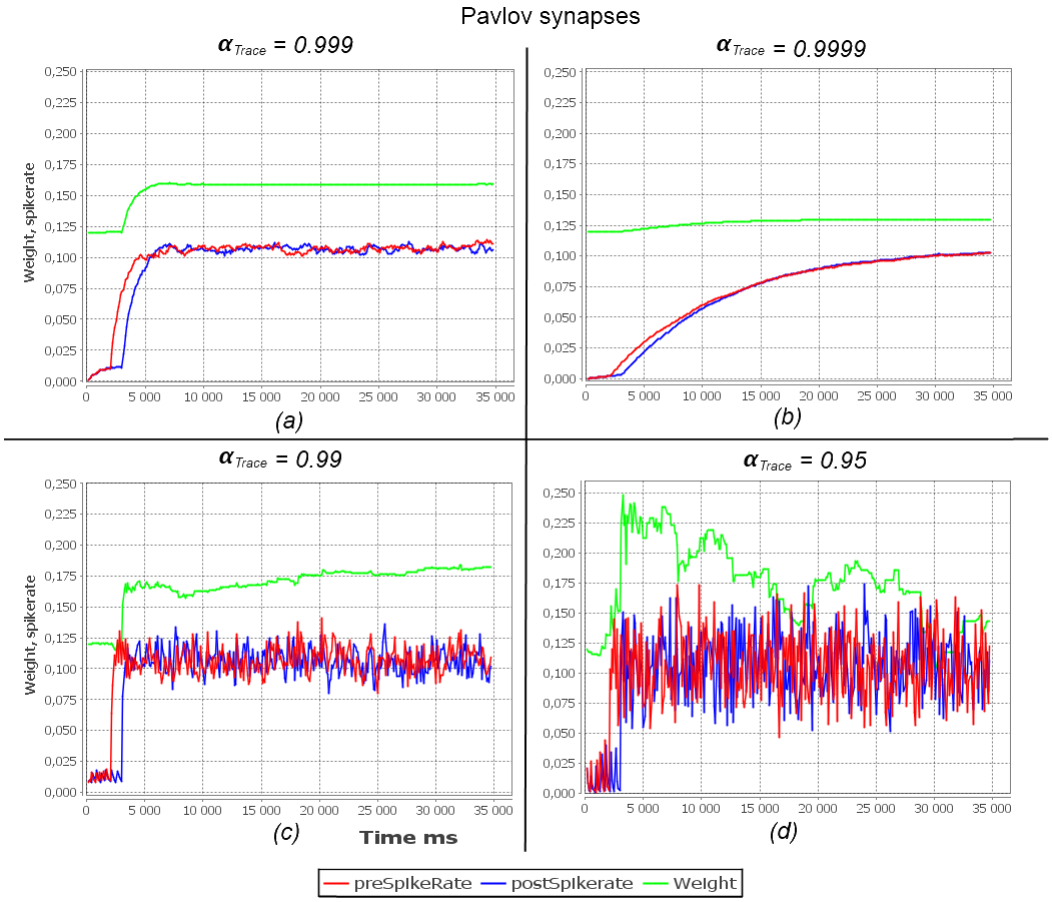

Figure 4.3: Varying α_{Trace}. The graphs show four different runs with the same settings apart form the α_{Trace} value. The spike rate is increased for the presynaptic- and postsynaptic neurons after 2000 and 3000 ms respectively. From this figure $\alpha_{Trace} = 0.99$ may not look so horrible looking at the weight, but if we recall the Skinner rule weight problem of Section 3.6.2 we can imagine that the problem would increase as the variation in spike rate increases.

If we on the other hand decrease the α_{Trace} to 0.99 or 0.95 we get other problems more similar to the problems of the two previous spike rate estimation methods. Figure 4.3(c) and (d) shows an estimate of the spike rate that is too local which is also a problem, we get large fluctuations and as we can see the learning rules will have problems updating the weight correctly.

4.4 Large History Window

I finally rethought the idea of keeping a long history of spikes. Earlier I thought that to get a good estimate of the current spike rate it would be best to weight the spikes in such a way that the last spike had a greater effect on the spike rate than the spike for say 1000 ms ago. But if I where to implement it that way I would have to loop the history through a sigmoid function (or similar) each time step for every neuron, which would be very time consuming[3] . Using a large window in large networks would also consume a lot of memory. But in my

[3]When testing the application using a weighted history of 1000 time steps, it ran over 50 times slower than normal. (e.g. a 2 second run would have $2000ms \cdot 1000history \cdot 50neurons = 1.0E8$ invocations of the weighting function which is a lot compared to the rest of

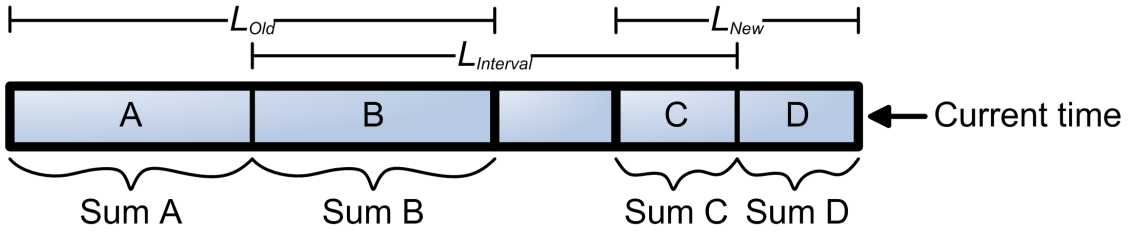

Figure 4.4: The definition of the spike history window used to calculate the change in spike rates.

case with a maximum of 102 neurons the memory issue was not a big problem.

The solution to the time problem was simply not to weight the spike history. When doing this I found that it worked really well. Despite my earlier concerns it appeared that a few seconds was not too much and I could still get a good estimate of the spike rate. The advantage of not weighting the input is that we do not have to loop the history at all. Instead we keep a sum of the history, and for each time step we simply add the newest value, and subtract the oldest value in the queue, which practically takes no time.

The main reason for changing from the trace based spike rate was the problem that the old trace based spike rate was not able to distinguish the order of the events as described in Section 3.6.3. When keeping a history this can be solved by using an old spike rate estimate for the neuron that is supposed to change its spike rate first. Figure 4.4 shows how the spike history window is defined by three parameters, L_{New}, L_{Old} and $L_{Interval}$. The next section will describe how these parameters are set for each of the learning rules, and then redefine the learning rule equations a bit.

4.4.1 New Spike Rate Estimates and Updated Learning Rule Equations

Note that neither of the Skinner, Pavlov or Hume rule is changed, only the way of implementing them. Based on the definition of the spike history window given in Figure 4.4 we get new estimates for the spike rate and the change in spike rate:

The current spike rate is defined as the average number of spikes over a given time frame.

$$S_{Window,t} = \frac{\sum D}{L_D} \tag{4.2}$$

where D is the spike history over the last L_D time steps. So in practice we estimate the spike rate at time: $currentTime - L_D/2$, meaning we have an offset of $L_D/2$.

The change in spike rate is defined a little bit different for the Skinner rule, and the Pavlov and Hume rules:

Skinner:

$$\triangle X_{Skinner,t} = \frac{2 \cdot (\sum D - \sum C)}{L_{New}} \tag{4.3}$$

$$\triangle Y_{Skinner,t} = \frac{2 \cdot (\sum B - \sum A)}{L_{Old}} \tag{4.4}$$

where $\triangle X_{Skinner,t}$ and $\triangle Y_{Skinner,t}$ are the spike rate changes in the pre- and postsynaptic neurons respectively. For the Skinner rule we notice that the spike rate of the presynaptic neuron is calculated based on the most recent history and the postsynaptic spike rate is calculated based on an older part of the history, this

the application.)

because we want the postsynaptic neuron to change before the presynaptic does. As we will soon see this is reversed for the Pavlov and Hume rules. But first the new Skinner rule equation:

$$\triangle W_{Skinner,t} = (-min(\triangle X_{Skinner,t}, 0) \cdot \triangle Y_{Skinner,t} - abs(\triangle Y_{Skinner,t}) \cdot W_t \cdot C_{Forget}) \cdot C_{Hok} \quad (4.5)$$

where W_t is the current weight of the synapse.

Pavlov & Hume:
For the Pavlov and Hume rule the change in spike rate is defined like this (still with reference to Figure 4.4):

$$\triangle X_{P\&H,t} = \frac{2 \cdot (\sum B - \sum A)}{L_{Old}} \quad (4.6)$$

$$\triangle Y_{P\&H,t} = \frac{2 \cdot (\sum D - \sum C)}{L_{New}} \quad (4.7)$$

The Pavlov equation then becomes:

$$\triangle W_{Pavlov,t} = max(\triangle X_{P\&H,t}, 0) \cdot (\triangle Y_{P\&H,t} - W_t \cdot C_{Forget}) \cdot C_{Hok} \quad (4.8)$$

And the Hume equation becomes :

$$\triangle W_{Hume,t} = -max(\triangle X_{P\&H,t}, 0) \cdot (\triangle Y_{P\&H,t} - W_t \cdot C_{Forget}) \cdot C_{Hok} \quad (4.9)$$

As we can see from these equations we do no longer utilize the trace functions of Equation 3.3 and 3.4. The next subsection deals with how the spike history window should be divided into A,B,C and D segments for the different rules.

4.4.2 Deciding the Three Window Parameters for the Three Rules

In one of my meetings with Hokland he shared his intuition that in the case of the Hume rule a presynaptic event can be followed by a postsynaptic event almost at the same time. For Pavlov synapses there have to be a little more time between the two events, and for Skinner synapses even more time between the events.

This led me to using these windows as a base:

Skinner window:

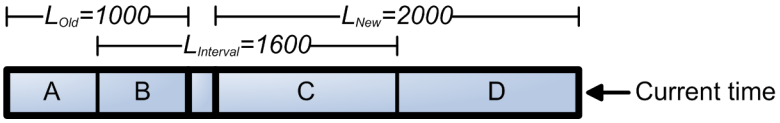

Figure 4.5: Skinner window, $L_{Old} = 1000$, $L_{Interval} = 1600$, $L_{New} = 2000$

Pavlov window:

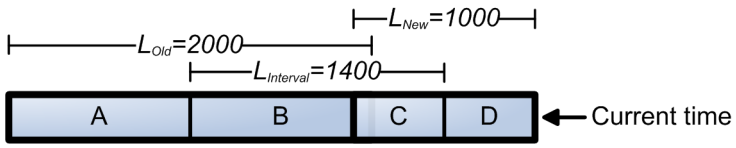

Figure 4.6: Pavlov window, $L_{Old} = 2000$, $L_{Interval} = 1400$, $L_{New} = 1000$. Note that in this case the C and B overlap a little.

Hume window:

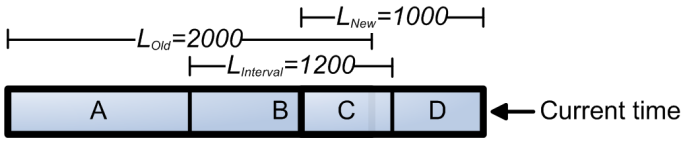

Figure 4.7: Hume window, $L_{Old} = 2000$, $L_{Interval} = 1200$, $L_{New} = 1000$. In this case B and C overlap even more.

4.4.3 Trace vs Large Window Based Spike Rate Estimation

The main reason for introducing the large window spike rate estimation methods was to deal with the "order of the events" problem described in Section 3.6.3. As we can see from Figure 4.8 this is solved by using the large window based methods.

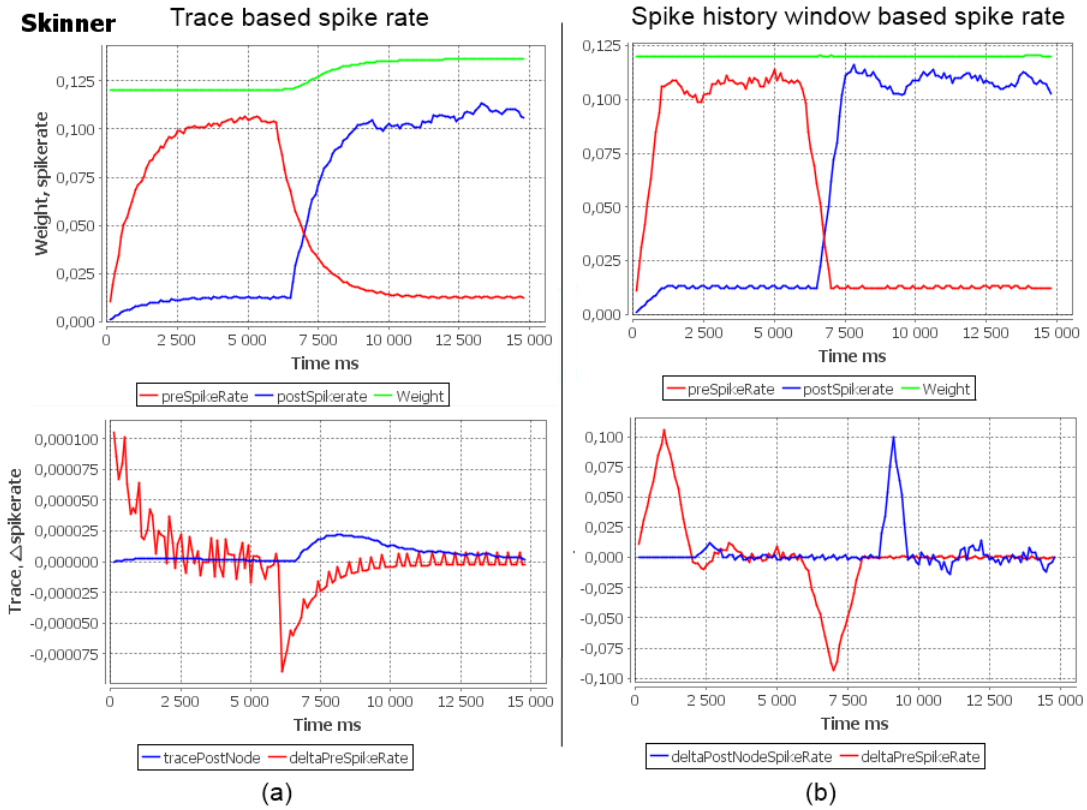

Figure 4.8: A Comparison of the trace based and the large window based spike rate estimations. Here we can see how the large window deals with "the order of the events" problem by using an older part of the history to calculate the postsynaptic spike rate change (bottom right graph). This way the weight is not wrongly updated when the presynaptic change occurs before the postsynaptic event (in the case of the Skinner rule).

Figure 4.9 shows another comparison of the trace based spike rate estimation (figure (a)) and the history window based spike rate estimation (figure (b)). As one might notice the figure shows the Skinner weight problem described in Section 3.6.2, and as we can see this problem is no longer visible when using the window based spike rate estimation method. The reason for this becomes clear when we look at the two bottom graphs,

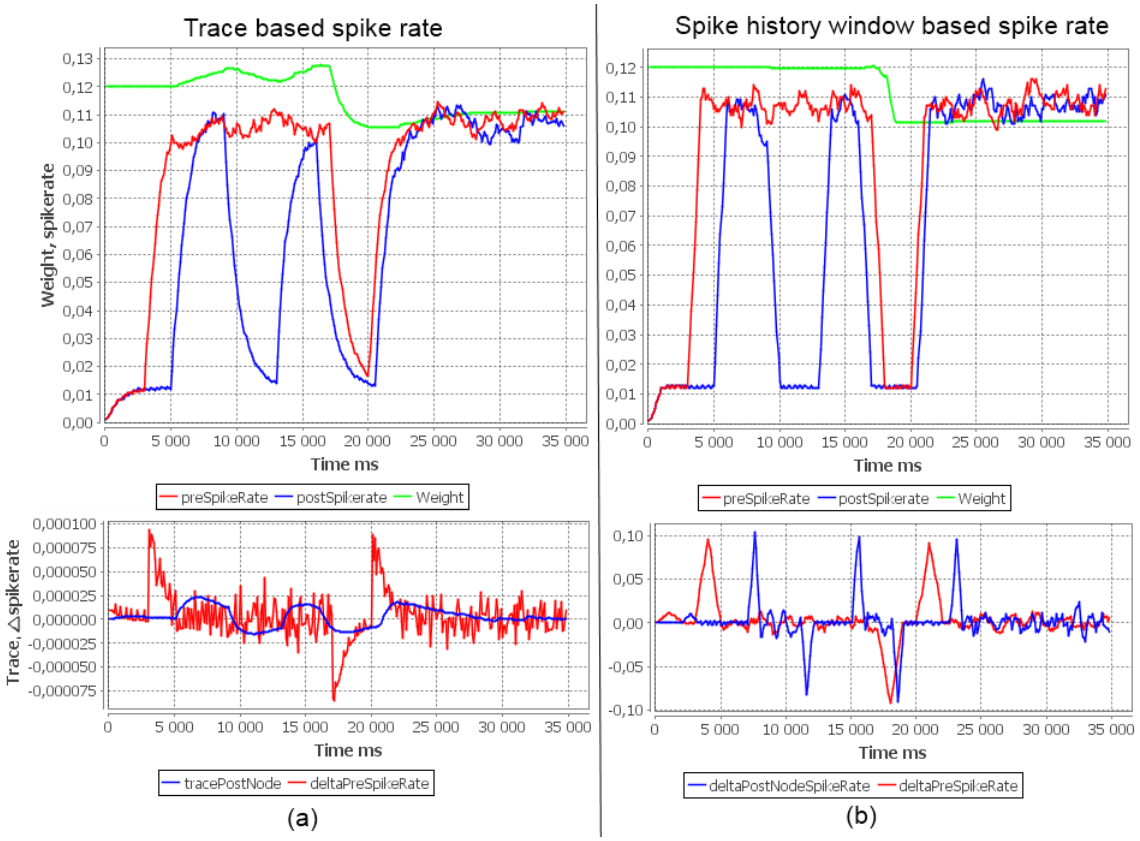

Figure 4.9: On the top left we got the trace based spike rate estimates according to Equation 4.1, the bottom left graph shows the change in presynaptic spike rate, and a trace of the change in the postsynaptic spike rate according to Equations 3.1 and 3.4. The top right side of the figure shows the spike rate estimates of the large window method according to Equation 4.2, and the bottom right figure shows the change in these spike rates according to Equations 4.3 and 4.4.

recalling that the Skinner rule updates its weight according to the postsynaptic activity (the blue graph) whenever the presynaptic activity drops (when the bottom red graph is below zero). As we can see the noise is greatly reduced when using the large window method for calculating the change in spike rate compared to the trace based method.

Now that we got the foundation in place it is time to see if the agent is able to learn the desired movement pattern which will help it get to the food source.

Chapter 5

Testing, Tuning and Results

This chapter describes my efforts to use the ANN I created to control the mechanical agent. I started out with four network topologies differing mainly in size (number of neurons). I tested each topology with a lot of different settings to see which of them performed best, and if any of them could self-organize.

5.1 The Testing and Tuning Procedure

The way I did the testing was to set the different parameters through the GUI and run the simulation. To help me figure out how to tune the different parameters as to obtain the desired behavior I mainly used four displays in addition to watching how the synaptic weights were updated.

The first display (Figure 5.1) is embedded in the GUI and shows an estimate of the current spike rate of every neuron. This is very helpful when tuning the A_{Syn} values of Equation 1.5. If for example the neurons in layer 5 are not firing at all I might increase the A_{Syn} values of the incoming synapses.

The second display is the spike rate and weight graphs shown several times earlier, first described in Section 2.5.1. These are very helpful in determining the different spike rate history window lengths, and to verify that the weight is updated when it is supposed to and that the learning constant is as wanted. It also helps making sure that the activity in the different neurons is varying given different input, i.e. it helps to tune the *a, b, c* and *d* parameters of Equation 2.1 in Section 2.1.2. If the agent moves closer to the goal and the spike rate of the horizontal distance neuron does not decrease I have to adjust some parameters to be able to record the need reduction as a clear spike rate reduction.

The third display shows the spike rate fed to the muscle model from the motor neurons, it also shows the length and velocity of all the muscles and the forces produced by the muscle model, see Figure 5.2.

In addition to this I will of course watch the behavior of the agent in the main window of the Webots application. I have equipped the agent with a marker so that it is easier to see where he has been, i.e. it will draw a black line as it moves. See Figure 5.3 for an example situation where the agent has moved a little towards the goal.

Based on the information given by the mentioned displays and the synaptic weights, I will tune the different parameters up and down to try to get the network to behave as wanted. After doing a lot of testing I have become familiar with what to tune in many situations, but it is rarely just to tune one parameter because they all affect each other.

Each of the next four sections will start with an image displaying the topology, the image will show some of the synapses, and the synapses that are not shown follow the same pattern as those that are displayed. The synapses are marked by a S, P, H or a C. This describes the synapse type where $S = Skinner$, $P = Pavlov$, $H = Hume$, $C = Constant/Static$. These are often followed by either a $'+'$ or a $'-'$ sign to specify whether they are excitatory or inhibitory respectively, if no sign is specified they can change from one to the other. Also initially I only used RES neurons since they are the most stable, but I will also try some setups with RS neurons.

The following four sections describe the different topologies and how the tuning of each of them went.

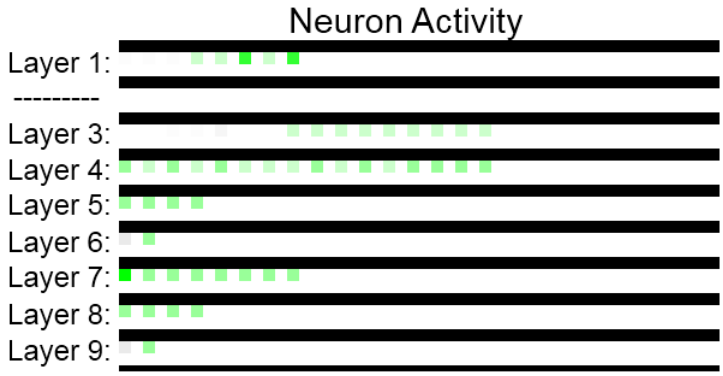

Figure 5.1: The GUI-display estimating the current spike rate of each neuron. The higher the spike rate, the darker green. Layer 1 is the motor neurons, layer 2 is not in use, layer 3 and 4 are the two different internal layers. Layer 5 receives motor force need input from layer 8, and layer 6 receives distance need input from layer 9, so layers 8 and 9 are used to transform the output from the agent to need input for layer 5 and 6. Layer 7 is the sensory input from the muscles (length and velocity).

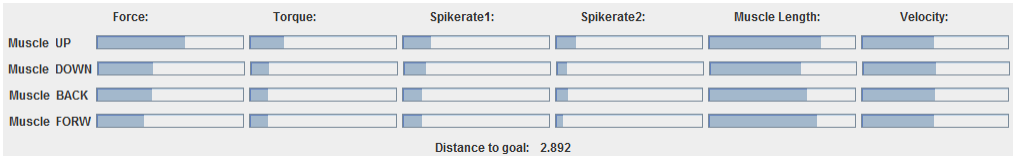

Figure 5.2: A display showing the spike rates of the two motor neurons associated with each muscle, the muscle length and velocity, and the forces these generate. This display is also updated in real time. Note that the values displayed are scaled to give more information, i.e. in reality the torque should have been ten times smaller to be in the same scale as the force but such small values would make it hard to study.

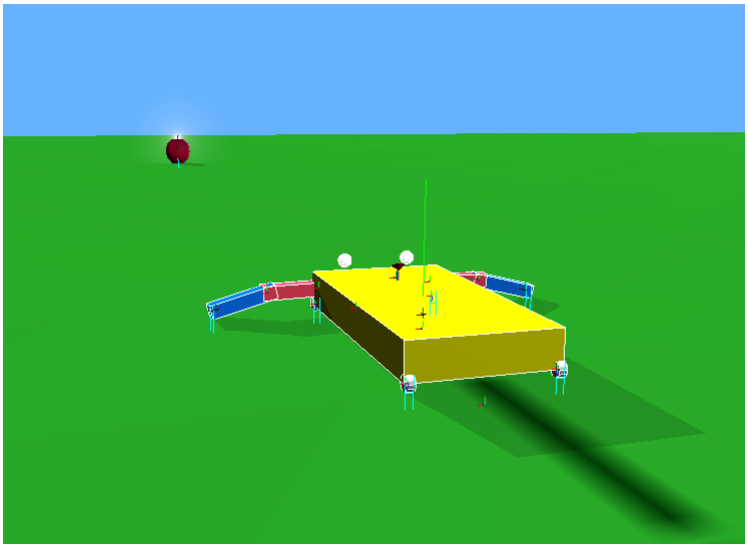

Figure 5.3: An example of the Webots main display, the teal colored vertical lines are the contact points between the agent and the ground, and the black line behind the agent marks the distance it has moved forward.

5.2 Topology 1: Minimal

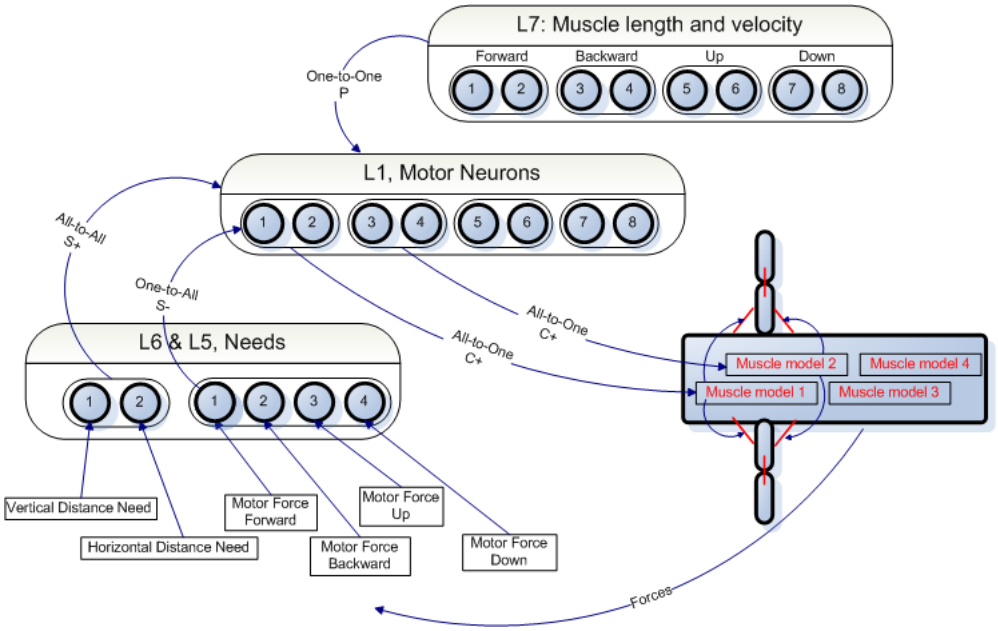

Figure 5.4: Minimal topology, the arrows represent synapses, none of the topology figures show all the synapses but the neurons without synapses follow the same pattern as the other neurons, e.g. in this case neuron 5.2 is connected to neuron 1.3 and 1.4. Note that when it says for example One-to-All like the S- connection from L5 to L1 that means that there is a connection from the selected neuron (neuron 5.1) to all the neurons in the group the arrow is pointing at, not necessarily the whole layer, in this case consisting of neuron 1.1 and 1.2.

The first topology is the simplest of them all. I did not have very big expectation for this topology due to the pattern that were to be learned without any internal neurons, 14 input neurons, and 8 output neurons. But it would be a good learning experience for me before starting to tune the larger networks, and of course there was always a chance that it could succeed.

For learning to happen from the need layers to the motor layer the muscle length have to change and stay changed for a while before the need goes down and not just go up and down all the time and thereby cancel the learning. It is a real challenge to get the timing right.

What is good with such a small network is that it is not that many variables to tune, so that makes it a bit easier to test.

5.2.1 Minimal Topology: Test Results

I tried some different setups with and without random weights, STDP, and random bursting, but as expected this topology did not lead to much self-organization. The agent would sometimes move a little back and forth but it was not able to learn anything significant based on the firing patterns of the different neurons, i.e. the time the agent did move a little forward and the distance need dropped a little, it was not preceded by a significant increase in the postsynaptic neuron. Typically in such a small network the increase in the motor neurons directly affect the need neurons, so that if the muscle length changes and the result is that agent pushes forward the need will go down about the same time but the weight is updated according to what happened say a second before the muscle changed, which in this case leads to a pretty random weight updating. I did not want to spend to much time and effort on this topology so I moved on to the next.

5.3 Topology 2: Small

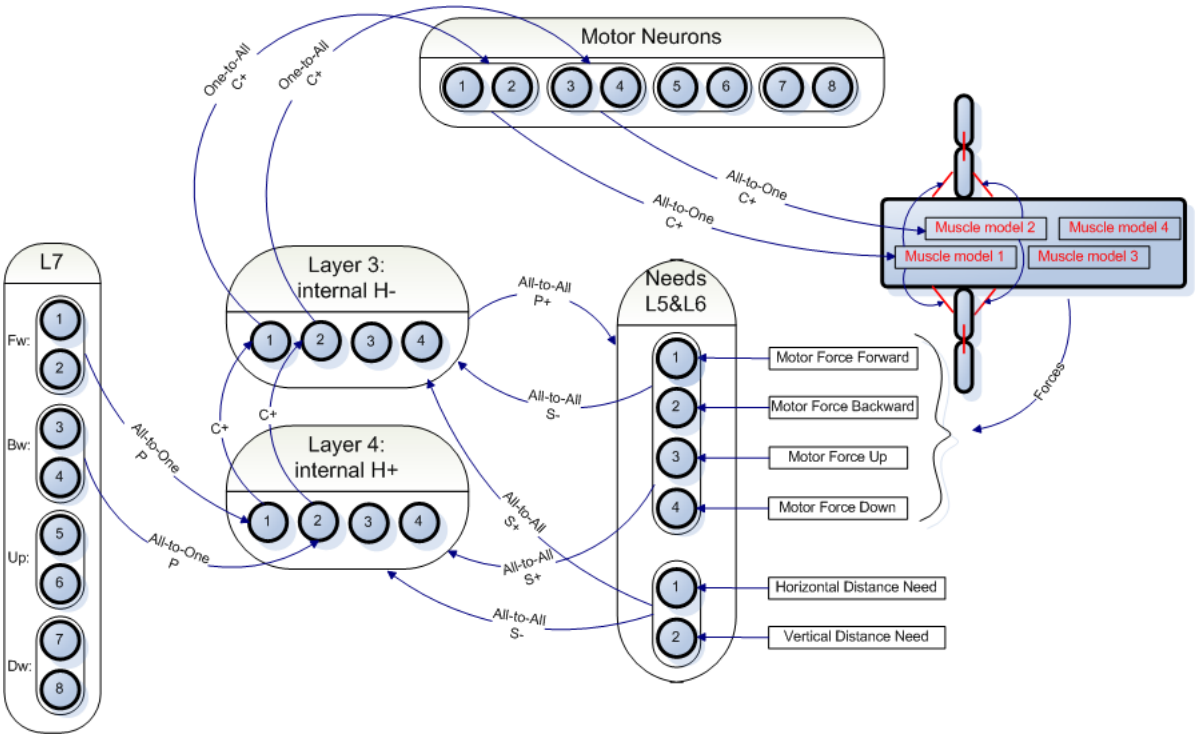

Figure 5.5: Shows the "Small" network topology. L7 contains muscle- length and velocity neuron pairs for each muscle. We have two internal layers, Layer 3 and Layer 4 which have internal all-to-all connections between the neurons (not drawn on the figure), layer 3 got inhibitory Hume synapses and layer 4 got excitatory Hume synapses. The other layers do not have internal synapses.

This topology moves a step up from the minimal topology by introducing two internal layers, each with four neurons internally connected by Hume synapses. Initially this setup looked a little bit more promising since the need neurons are not directly connected to the motor neurons but to two internal layers. The synaptic weights are initially set to 0.5 (-0.5 for inhibitory synapses) for this topology.

There are several reasons for testing with small topologies first, not necessarily because I believe they are going to self-organize in the desired way, but it is much easier to get familiar with the network and to know what effects the different connections have and thereby get a better intuition of how to tune larger networks with a similar structure. Table 5.1 shows the initial parameter set used for this topology.

5.3.1 Small Topology: Tuning and Test Results

The first run with these settings resulted in a little movement back and forth, see Figure 5.6. The spike rate of the neurons in layer 3 and 4 would typically increase and decrease very often and it looked pretty random when it hit the decrease of the need neurons. I increased the window size to {1600,2100,2600} hoping that the spike rate would not vary as much. It was also generally high forces on all of the motors, in other words they were working against each other. After some tuning and testing this led me to change the synapses from the motor force need neurons so that so that neuron 5.1 had an excitatory synapse S+ to neuron 3.1 and inhibitory synapses S- to 3.2-3.4 (I also used the same pattern for neurons 5.2-5.4). Then I increased the A_{Syn} value of the internal synapses in layer 3 to increase the competition, which resulted in a generally lower activity in the network, so

Parameter	Value	Parameter	Value
SynBeta:	1	A_{Syn} L3	1.0
LearningConstant:	100	A_{Syn} L3->L5	0.5
ForgettingConstant:	1.0E-5	A_{Syn} L3->L6	0.1
STDPLearningConstant:	0.0	A_{Syn} L3->Motor	20.0
WeightUpdateFreq:	100	A_{Syn} L4	1.0
Random burst:	-	A_{Syn} L4->L3	1.0
Random weights:	-	A_{Syn} L5->L3	0.5
Initial Weights:	$\pm$0.5	A_{Syn} L5->L4	0.5
Internal delays:	0	A_{Syn} L6->L3	1.0
Inter layer delays:	0	A_{Syn} L6->L4	1.0
SpikeRateType:	LargeWindow	A_{Syn} L7->L4	5.0
WindowSkinner{L1,Interval,L2}:	{1000,1500,2000}	A_{Syn} MotorForces->L5	5.0
WindowPavlov{L1,Interval,L2}:	{2000,1200,1000}	A_{Syn} DistanceNeeds->L6	5.0
WindowHume{L1,Interval,L2}:	{2000,1000,1000}		

Table 5.1: The initial parameter set of the "Small" topology. For an explanation of the parameters see Appendix A.1.

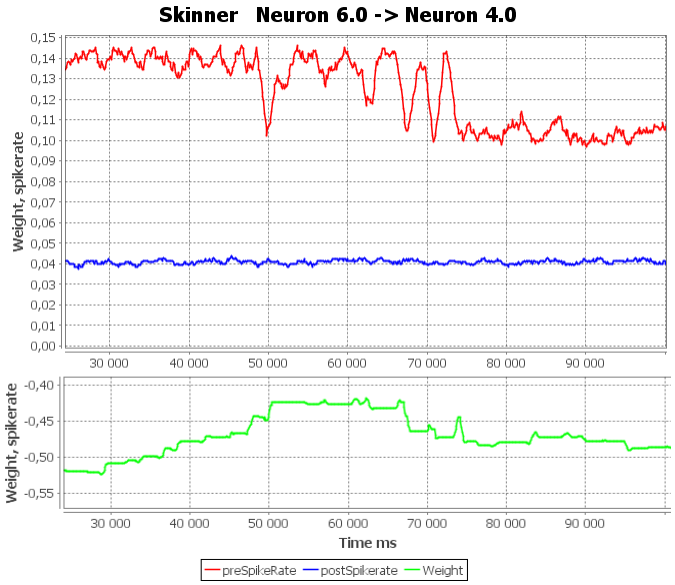

Figure 5.6: Shows the spike rates of need neuron 6.0 and internal neuron 4.0 with the initial settings specified in Table 5.1, where 6.0 is the horizontal distance need. In this case the agent moved a little back and forth until about 75 000 ms then it basically held its position. If we look at the weight we can see that it moves a little up and down. It is hard to say if this is just random or if it is helping the network learn, i.e. it is not sure neuron 4.0 had much to do with the forward movement. Generally it is very hard to say anything for certain just by looking at one synapse. As we can see there are not very big changes in the postsynaptic spike rate. This may be an indicator that it would be wise to increase the effect the muscle length and velocity neurons have on layer 4, and also the effect of the motor force need neurons from layer 5 to layer 4, and decrease the internal connections of layer 4.

I increased the influence from layer 4 a bit. I would also increase the A_{Syn}L7->L4 and A_{Syn}L5->L4 some to get bigger variations in the spike rate of layer 4.

The robot would still move a little back and forth, if we look at Figure 5.7 we can see that the learning does not have a great effect the first 50 000 ms since the need generally does not go down, but still it seems that the small drops in a bigger degree than before is preceded by increased spike rate in the postsynaptic neuron so that the weight are generally increasing.

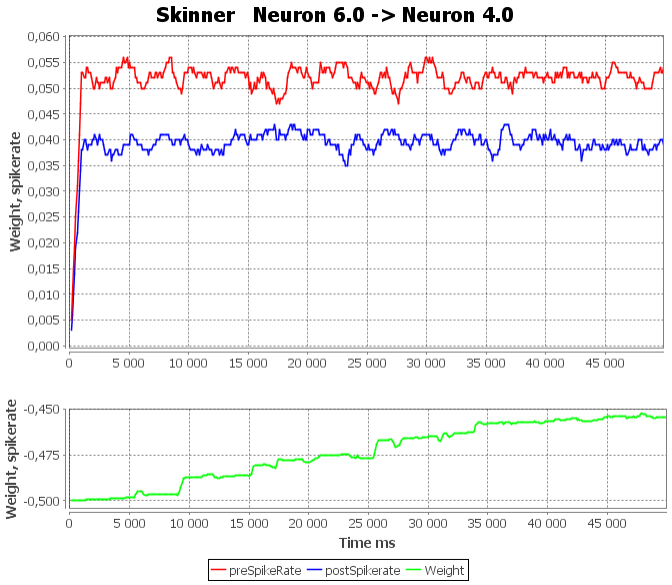

Figure 5.7: An example of spike rates and weight between neuron 6.0 and 4.0. The agent was moving a bit back and forth, and the weight increased slowly, but it did not lead to any big changes in behavior. The agent generally moved to little back and forth and the activity of neuron 4.0 seemed to fluctuate so that the weight did not move much in any direction (note that the scale on the weight plot is smaller than for Figure 5.6). I could increase the learning constant but that would lead to much to big changes when the need suddenly dropped a bit more while the postsynaptic spike rate changed in one direction as well.

After some more tuning and testing it seemed like the agent eventually learned to avoid using up and down muscles at the same time as well as the backward and forward muscles, but the order was pretty random so it would not move much forward, and after a while it usually got stuck pushing down and backward. I tried adding some STDP learning since that can lower the weights between neurons that spike to often. However this did not help. Since the spike rate was not high enough it actually worked against its purpose.

I also tested a structure with S+ synapses from layer 6 to layer 4, and S- synapses from layer 5 to layer 3, and no connections from layer 6 to layer 3 and layer 5 to layer 4, but it did not seem to improve the learning and movement much. Using Pavlov synapses between layer 3 and layer 4 so that layer 3 could learn to anticipate the activity of layer 4 did not help either. Figure 5.8 shows a successful weight update. The problem is that in 200 000 ms it was the only clear spike rate drop in the distance need neuron.

Summary:

Generally it seemed like the network was not able to learn to isolate the different muscles without ending up stuck in one position, i.e. always pushing down and back. Hopefully with a larger network this issue could be solved. A larger network may change many weights a little, instead of few a lot, and I believe that can be a good thing when trying to learn a sequence of actions. The topology and settings I ended up with after a lot of tuning back and forth can be seen in Figure 5.9 and Table 5.2.

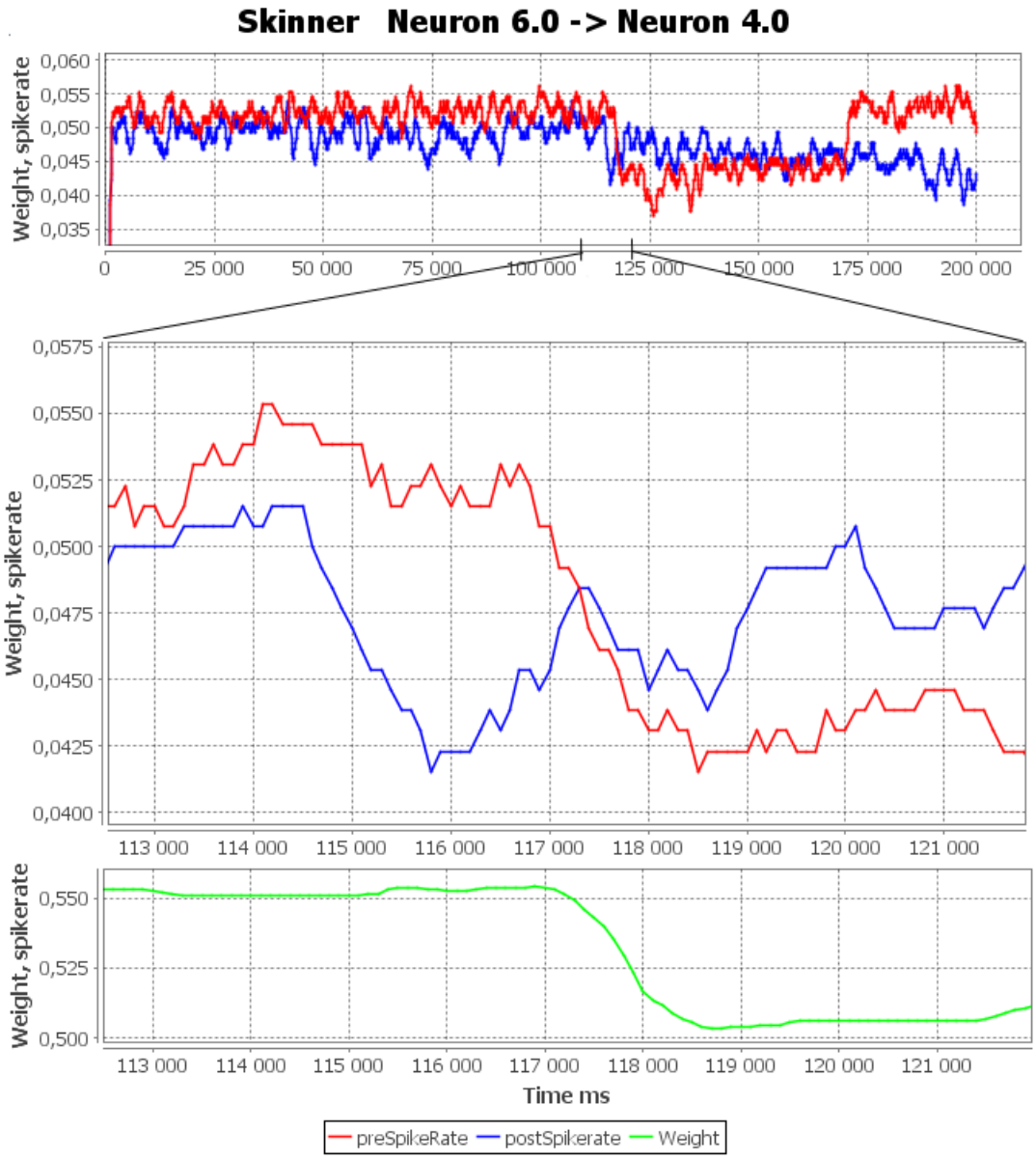

Figure 5.8: The top graph shows the spike rate of horizontal distance need neuron 6.0 and neuron 4.0, the middle graph shows an enlarged window from 112 500 to 122 000 ms, and the bottom graph shows how the weight is updated in that window.

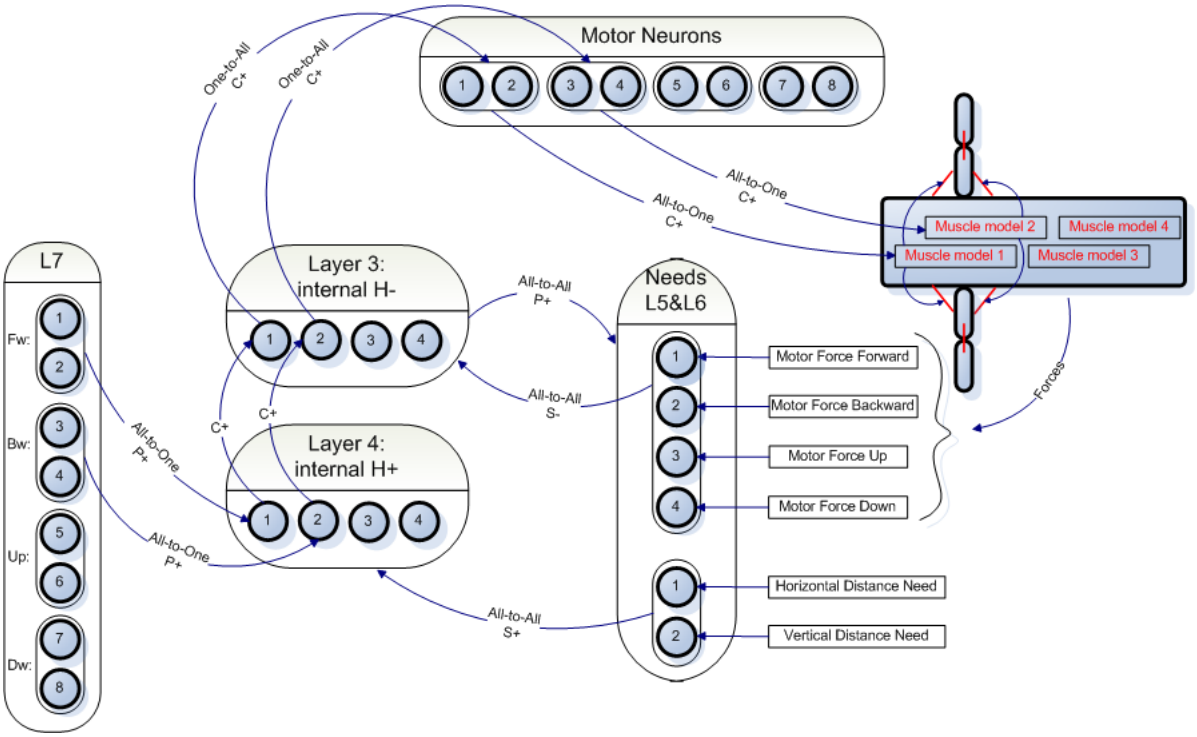

Figure 5.9: Shows the best topology I found with only 8 internal neurons. Note that I changed the Pavlov synapses from layer 7 to layer 4 to be P+ rather than P, since when using P synapses the weight often became large inhibitory and that led the postsynaptic neuron of layer 4 to completely stop firing.

Parameter	Value	Parameter	Value
SynBeta:	1	AsynL3	*4.0*
LearningConstant:	100	AsynL3->L5	0.1
ForgettingConstant:	1.0E-5	AsynL3->L6	0.1
STDPLearningConstant:	0.0	AsynL3->Motor	40.0
WeightUpdateFreq:	100	AsynL4	1.0
Random burst:	-	AsynL4->L3	*2.0*
Random weights:	-	AsynL5->L3	*0.4*
Initial Weights:	$\pm$0.5	AsynL5->L4	-
Internal delays:	*5*	AsynL6->L3	-
Inter layer delays:	*5*	AsynL6->L4	*1.0*
SpikeRateType:	LargeWindow	AsynL7->L4	*4.0*
WindowSkinner{L1,Interval,L2}:	*{1600,2100,2600}*	MotorForces->L5	*1.0*
WindowPavlov{L1,Interval,L2}:	*{2600,1800,1600}*	DistanceNeeds->L6	*1.0*
WindowHume{L1,Interval,L2}:	*{2600,1500,1600}*		

Table 5.2: The final parameter set of the "Small" topology. The parameters written in green is parameters that have been increased from the initial settings, and the red values marks a decrease from the initial settings.

5.4 Topology 3: Medium

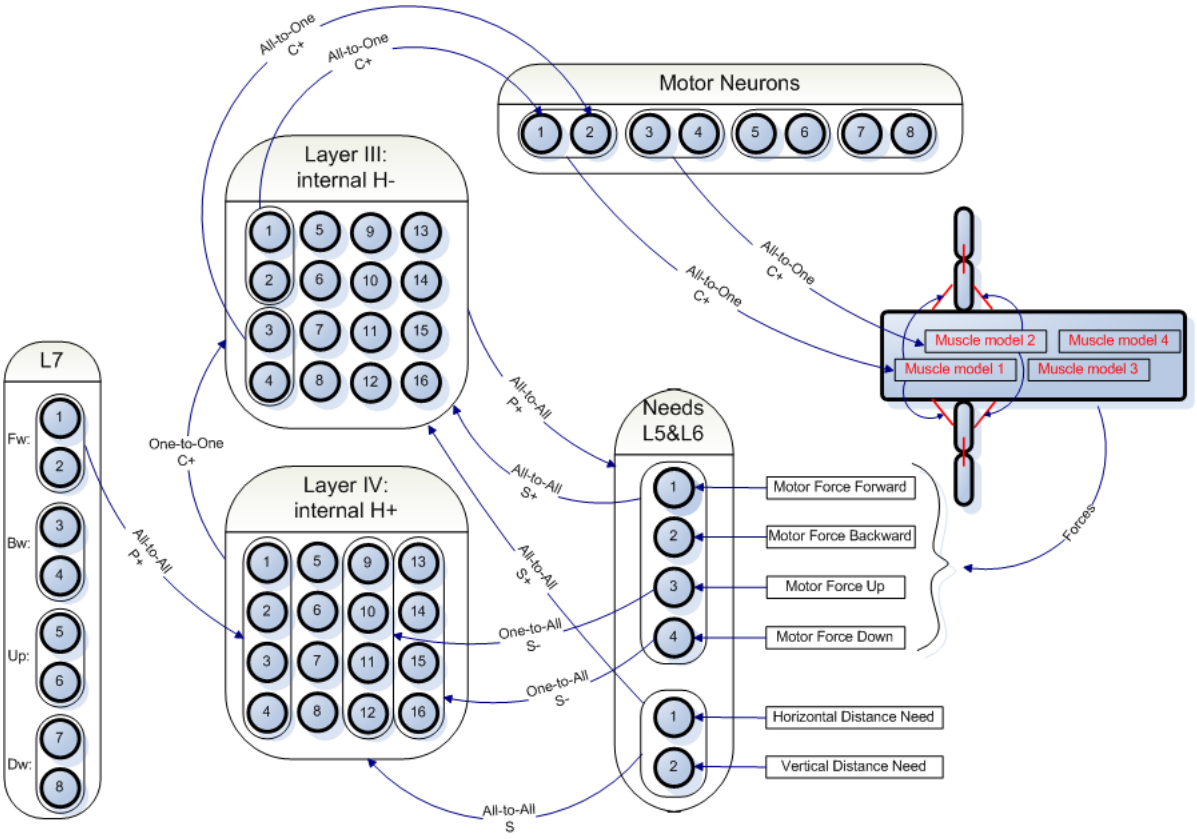

Figure 5.10: Topology 3, Medium. This topology uses four internal neurons for each motor group. It has a total of 32 internal neurons, 14 input neurons, and 8 output neurons.

This was the topology I had highest expectations for, with 2x16 internal neurons I was hoping that the network could be able to form some "plans", some internal patterns activated in such a way to result in the desired movement pattern. I believe it is necessary to form some partially overlapping patterns to be able to learn the wanted movement pattern and the timing necessary to produce forward movement. It is hard for me to imagine that only four neurons can learn to work together and be switched on and off by each other as to produce forward movement. When dealing with muscle forces and spike rates it is harder than it looks to control the agent manually by setting the different spike rates. When only using 4x2 internal neurons, each neuron would have a great influence on the network. With 16x2 internal neurons each neuron would have less influence but together it is easier to imagine that there could arise self-organizing patterns. Figure 5.10 shows the initial topology and Table 5.3 shows the initial parameter set.

5.4.1 Medium Topology: Tuning and Test Results

The initial settings worked for some minutes, the agent started with some random back and forth movement, but the total activity of the network kept increasing until the forces got out of hand and the agent started to act really unstable. So either the learning constants were to high, or the A_{Syn} values of the excitatory and the inhibitory synapses were unbalanced. When looking at the synaptic weights after stopping the simulation, they had not increase too much so they should not be the main problem. But as I was looking at the weights it became clear

Parameter	Value	Parameter	Value
SynBeta:	1	A_{Syn} L3	8.0
LearningConstant:	50	A_{Syn} L3->L5	1.0
ForgettingConstant:	1.0E-5	A_{Syn} L3->L6	1.0
STDPLearningConstant:	0.0	A_{Syn} L3->Motor	10.0
WeightUpdateFreq:	100	A_{Syn} L4	4.0
Random burst:	-	A_{Syn} L4->L3	1.0
Random weights:	-	A_{Syn} L5->L3	1.0
Initial Weights:	$\pm$0.05	A_{Syn} L5->L4	10.0
Internal delays:	5	A_{Syn} L6->L3	2.0
Inter layer delays:	5	A_{Syn} L6->L4	9.0
SpikeRateType:	LargeWindow	A_{Syn} L7->L4	8.0
WindowSkinner{L1,Interval,L2}:	{2000,2500,3000}	A_{Syn} MotorForces->L5	2.0
WindowPavlov{L1,Interval,L2}:	{3000,2100,2000}	A_{Syn} DistanceNeeds->L6	5.0
WindowHume{L1,Interval,L2}:	{3000,1800,2000}		

Table 5.3: The initial parameter set of the "Medium" topology. As we can see I started out with really high values for some of the A_{Syn} parameters, in particular I wanted high competition in layer 3, and a varying spike rate in layer 4. For an explanation of the parameters see Appendix A.1.

```
Neuron 5.1 ------- S- -0.028533333333333407 ------>  Neuron 4.4, Asyn:10.0, Delay: 5
Neuron 5.1 ------- S- -0.028533333333333407 ------>  Neuron 4.5, Asyn:10.0, Delay: 5
Neuron 5.1 ------- S- -0.028533333333333407 ------>  Neuron 4.6, Asyn:10.0, Delay: 5
Neuron 5.1 ------- S- -0.028533333333333407 ------>  Neuron 4.7, Asyn:10.0, Delay: 5
Neuron 6.0 ------- S   0.0297333333333334   ------>  Neuron 4.4, Asyn: 9.0, Delay: 5
Neuron 6.0 ------- S   0.0297333333333334   ------>  Neuron 4.5, Asyn: 9.0, Delay: 5
Neuron 6.0 ------- S   0.0297333333333334   ------>  Neuron 4.6, Asyn: 9.0, Delay: 5
Neuron 6.0 ------- S   0.0297333333333334   ------>  Neuron 4.7, Asyn: 9.0, Delay: 5
Neuron 7.4 ------- P+  0.02643333333333336  ------>  Neuron 4.8, Asyn: 8.0, Delay: 5
Neuron 7.4 ------- P+  0.02643333333333336  ------>  Neuron 4.9, Asyn: 8.0, Delay: 5
Neuron 7.4 ------- P+  0.02643333333333336  ------>  Neuron 4.10,Asyn: 8.0, Delay: 5
Neuron 7.4 ------- P+  0.02643333333333336  ------>  Neuron 4.11,Asyn: 8.0, Delay: 5
```

Figure 5.11: Some of the synapses from layer 5, 6 and 7 to layer 4, as we can see all the weights from the different groups are identical, which is a clear indicator that the post synaptic neurons always fire at exactly the same time.

that all the neurons in each of the subgroups in layer 4 fired in the exact same pattern, i.e. all the weights was updated the same way, see Figure 5.11.

The reason for this is of course that they all get the same input, so to deal with this I either had to introduce random initial weights or I had to change the topology. I started with changing to random weights to see if that was enough. Figure 5.12 shows a comparison of the initial weight of 0.05, and random weights in the range (0, 0.1).

When running some more tests I actually got a very promising run, a recording of it can be seen by following the link in the footnote below[1]. As we can see from the video the agent started good, but after a while it became more and more unstable and finally got stuck using only the forward and down muscles. So then it was back to the tuning bench.

I then decided to alter the topology a little by changing the connections from layer 5 to layer 4 into All-to-All connections hoping that it would lead to more overlap so that the agent would not get stuck in one position. When this only led to a minimal movement since all motors generated about the same force I decided to let the neurons in each subgroup in layer 3 have internal C+ synapses instead of H- synapses so that neurons activating the same motor neuron should not inhibit each other (there would still be H- synapses between neurons in the different subgroups). When testing with this setup the agent was a bit more stable it would move a little back

[1] A promising run, YouTube clip: http://www.youtube.com/watch?v=UTMYHfJDClE

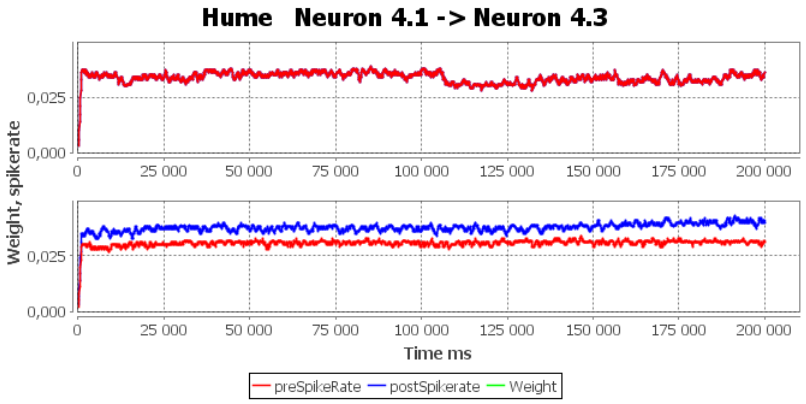

Figure 5.12: The top figure shows the spike rate of neuron 4.1 and 4.3 with initial weights of 0.05 as we can see they are completely overlapping. The bottom figure shows the spike rate of the same neurons with random initial weights in the range (0, 0.1), which separates the two spike rates.

and forth but after a while it would get stuck because a few weights had increased a lot while others had dropped to zero.

After a lot of tuning and testing, introducing random bursting [2], using STDP learning, and varying the initial weights, I still could not find a very good parameter set. It seemed like after running the simulation long enough the agent would most of the time get stuck in one position. A few times it would not get stuck but the forces would start alternating very frequently and the agent would "explode", i.e. the arms would jump out of joint and the agent would bounce around[3]. And the times none of these two happened the agent would not move much at all. [4]

But I did not want to give up quite yet. As we can see from Figure 5.12 the variation in spike rate in layer 4 is not very big, even thou the image is lying a bit due to the scale on both the x and y axis. I had already tuned the input from layer 7 to vary a lot for small differences in length and velocity, so I decided to try using RS neurons in layer four instead of the current RES neurons, since RS neurons are more responsive to input. This did not work at all since the RS neurons were much to unstable either they fired to much or not at all. To deal with this I tried to use a mixture of 20% RS neurons and 80% RES neurons. In practice I did this by adding four RS neurons to both layer 3 and 4, one for each muscle. This setup gave the best result so far. The agent would move back and forth without getting stuck, the only thing that were missing was a clear movement pattern, the agent did not seem to learn when to lift his arms. It could move slowly forward for a while and then making some wrong moves and move right back to the beginning. A recording of this movement is shown in the link in the following footnote[5]. I watched the agent for 15 minutes but I still could not see any pattern emerging. I looked at some of the weights and thought they changed a bit too slow so I increased the learning constant hoping that I could get some better results, but it did not seem to help the agent would just get stuck a little more often. So I decided to let it run over night, but there was still no apparent self-organization.

The problem probably was not that the learning constant was to small but that there was not a consistent connection between the activity of the neurons in layer 3 and 4, and the movement of the agent, i.e. there had not been formed any distinct firing pattern internally in those layers. If for example the first time the need dropped, neuron 4.1 had previously increased its spike rate, while neuron 4.2 had decreased its spike rate, then the next time the need dropped the situation was reversed and the learning would cancel. This demonstrates an

[2]The random bursting feature is designed so that if a random value in the range (0, 1) is below the specified value in the GUI the given neuron will fire once every ten time steps for 1000 time steps. So that a neuron that is not firing at all would then fire at 100 Hz for a second, so the value specified in the GUI should be very small, I generally used a value of *0.0001*. Also note that the estimated spike rate would not increase to 100Hz since it is calculated over a larger window.

[3]A short recording of such an event can be seen at here: http://www.youtube.com/watch?v=JaBvRB66ZG0

[4] *"There's a fine line between genius and insanity."* - Oscar Levant, I am starting to think that is true for robots too.

[5]The best result for the medium topology: http://www.youtube.com/watch?v=GKxZP1II3QQ

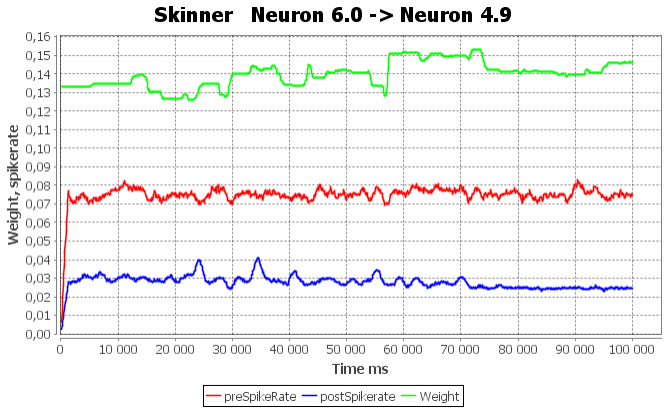

Figure 5.13: Lack of consistency. The figure shows the spike rate of neuron 6.0 which is the horizontal distance need, and neuron 4.9 which is associated with the backward muscle. As we can see the need goes fairly much up and down, i.e. the agent moves back and forth, but judging by the weight update it seems to be pretty random what happens in neuron 4.9 prior to the need reduction.

advantage of small networks where it is only one internal neuron associated with each motor neuron. Figure 5.13 shows an example of the spike rates and corresponding weight of a distance need neuron and one of the neurons in layer 4.

Another thing that could be wrong is the timing, i.e. the length of the different spike rate window sections, but when looking at the spike rate graphs of some of the neurons it is hard to see any patterns that could guide me in setting the window parameters.

I finally decided to move on to try an even larger topology hoping for some luck there. Figure 5.14 and Table 5.4 shows the topology and parameter set I ended up with for the medium topology.

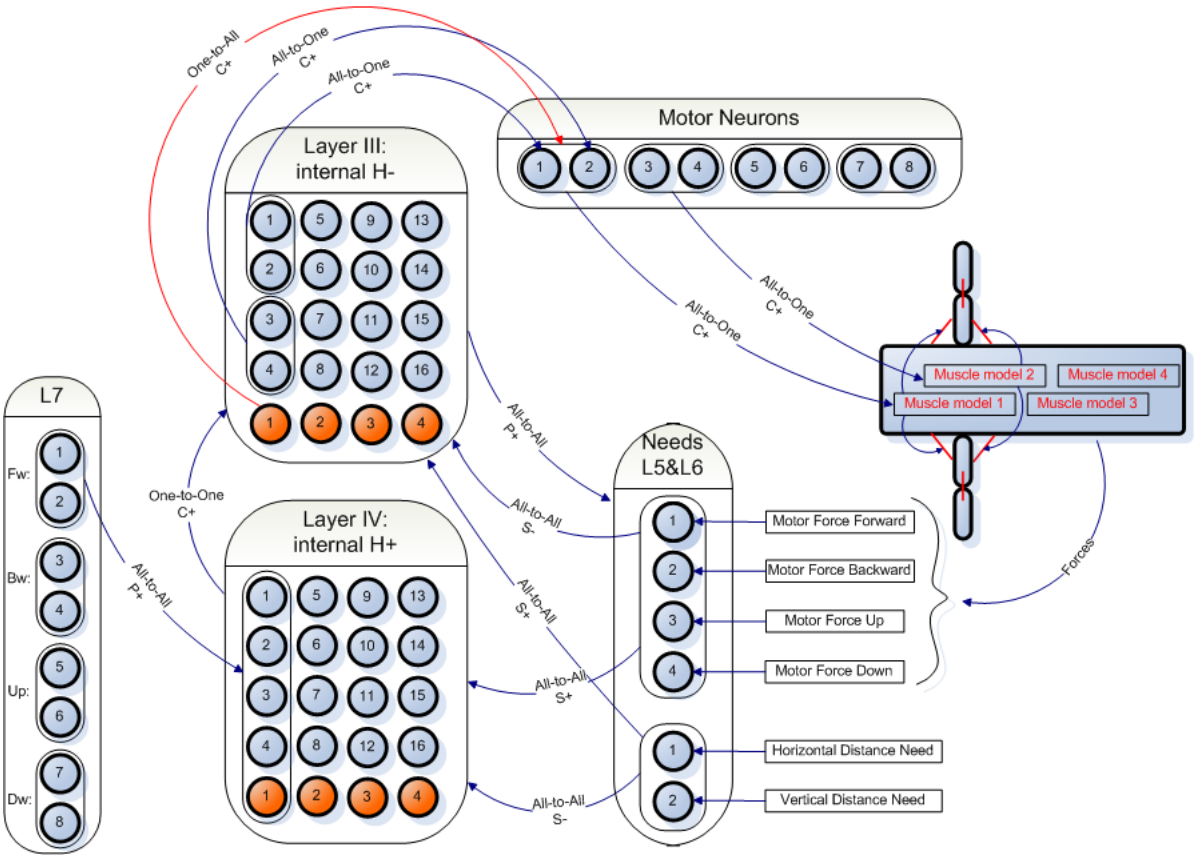

Figure 5.14: The medium topology that gave the best results, note that there are C+ synapses between each of the neurons in each subgroup in layer 3 in stead of H- , but there is still H- between the different subgroups. The orange neurons represent the RS neurons.

Parameter	Value	Parameter	Value
SynBeta:	1	A_{Syn} L3	3.5
LearningConstant:	80	A_{Syn} L3->L5	0.005
ForgettingConstant:	1E-6	A_{Syn} L3->L6	0.005
STDPLearningConstant:	0	A_{Syn} L3->Motor	0.5
WeightUpdateFreq:	100	A_{Syn} L4	0.05
Random burst:	-	A_{Syn} L4->L3	5.0
Random weights:	1.0	A_{Syn} L5->L3	5.0
Initial Weights:	-	A_{Syn} L5->L4	4.0
Internal delays:	5	A_{Syn} L6->L3	2.0
Inter layer delays:	5	A_{Syn} L6->L4	4.0
SpikeRateType:	LargeWindow	A_{Syn} L7->L4	5.0
WindowSkinner{L1,Interval,L2}:	*{1600,2100,2600}*	A_{Syn} MotorForces->L5	2.0
WindowPavlov{L1,Interval,L2}:	*{2600,1800,1600}*	A_{Syn} DistanceNeeds->L6	2.0
WindowHume{L1,Interval,L2}:	*{2600,1500,1600}*	*20% RS neurons in*	*L3&L4*

Table 5.4: The best parameter set I found for the “Medium” topology.

5.5 Topology 4: Large

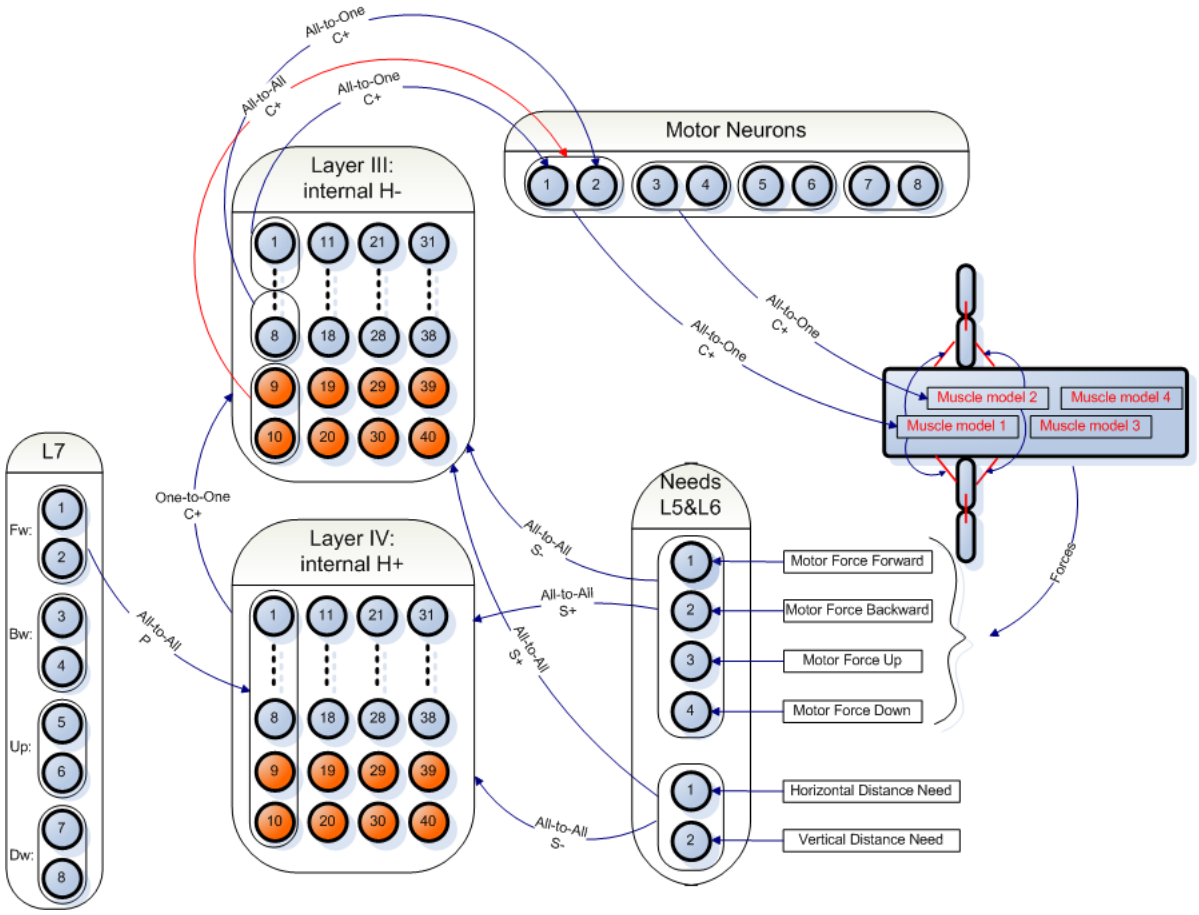

Figure 5.15: Large topology, each motor neuron pair gets input from 10 different neurons. A total of 80 internal neurons, 14 input neurons, and 8 output neurons, and about 4000 synapses. I tried using a similar setup for this topology as for the best topology in last section.

What I hope will be an advantage of this topology is that one neuron will not affect another neuron much since each neuron now get input from a lot of other neurons. I hope this will reduce the problem of the agent getting stuck after a while, even with a relatively large learning constant. When using such a large topology with random initial weights the chance of all motors receiving the same input is also greatly reduced. A problem with this topology is that one may get very different behaviors using the same parameter set, but hopefully the behaviors that led to some back and forth movement will update the weights so that they will become more and more similar as time passes. The main idea behind this topology is that small changes to many synapses are better than large changes to a few synapses. A larger network will also lead to a more stable behavior and smoother movement. For example if one of the neurons in layer 3 of the small topology (Section 5.3) stopped firing that would mean no force to the associated muscle. In the large network topology on the other hand each motor neuron group receives input from ten other neurons so it would only decrease the muscle activity a little bit if one of the neurons stopped firing.

Parameter	Value	Parameter	Value
SynBeta:	1	A_{Syn} L3	1.0
LearningConstant:	10	A_{Syn} L3->L5	0.005
ForgettingConstant:	1.0E-5	A_{Syn} L3->L6	0.005
STDPLearningConstant:	1.0E-6	A_{Syn} L3->Motor	1.0
WeightUpdateFreq:	100	A_{Syn} L4	0.1
Random burst:	0.0001	A_{Syn} L4->L3	1.0
Random weights:	1.0	A_{Syn} L5->L3	3.0
Initial Weights:	-	A_{Syn} L5->L4	2.0
Internal delays:	5	A_{Syn} L6->L3	1
Inter layer delays:	5	A_{Syn} L6->L4	1
SpikeRateType:	LargeWindow	A_{Syn} L7->L4	2.0
WindowSkinner{L1,Interval,L2}:	{1600,2100,2600}	A_{Syn} MotorForces->L5	2.0
WindowPavlov{L1,Interval,L2}:	{2600,1800,1600}	A_{Syn} DistanceNeeds->L6	5.0
WindowHume{L1,Interval,L2}:	{2600,1600,1600}	*20% RS neurons in*	*L3&L4*

Table 5.5: The initial parameter set of the Large topology. I have lowered the learning constant quite a bit due to the total number of synapses has increased a lot, and in hope that there would be many small updates rather than a few large.

5.5.1 Large Topology: Tuning and Test Results

When starting the simulation the agent had the desired back and forth movement so there was a basis for learning, but when looking at some of the weights after a couple of minutes there were almost no learning indicating that the learning constant was too low. After increasing the learning constant and tuning some of the A_{Syn} values the learning could be clearly seen, Figure 5.16 shows an example of one of the synapses.

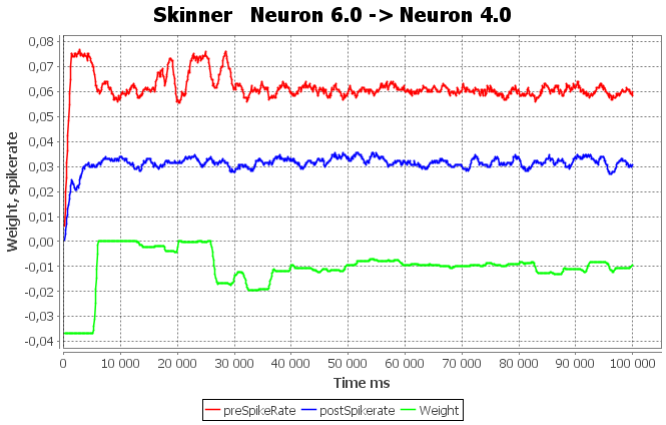

Figure 5.16: Shows an example of Skinner learning the first 100 seconds.

As expected this topology resulted in slower and less movement since one have to change many more neurons before there is a clear change in behavior. I wanted some more action so I amplified the differences in the motor neuron output by changing the type of the motor neurons from RES to RS neurons. As mentioned earlier the RS neurons are more responsive to input but need more activation to get started. So I adapted the A_{Syn} value from layer 3 to the motor neurons to obtain a good activity. The result was more variation in the activity in the motor neurons, but the activity would not alternate more between the neurons so basically it led to forces that were not working as much against each other, but the agent would get stuck pretty fast. I then increased the effect of the muscle- length and velocity neurons since they work against the current position of the arm, i.e. if a muscle is long it will spike more which in turn will contribute to a muscle contraction. But this

is a fine balance since one do not want the neurons of layer 7 to have to much influence, if this happens the need neurons that are supposed to be the driving force of the network would lose much of their effect.

With such a large topology it is hard to see exactly what is wrong in the firing patterns, so most of the tuning here was based on the experience from the smaller topologies. I would look at some neurons and how the synaptic weight was updated as in Figure 5.16, but a single synapse will not reveal much about the whole. After some more testing and tuning I kept running into the same problems as for the smaller topologies. I did not spend as much time on this topology since I did not get very promising results and my main focus was on the "medium" topology. A short recording of a run can be seen following the link in the footnote[6]The topology and settings I ended up with can be seen in Figure 5.17 and Table 5.6.

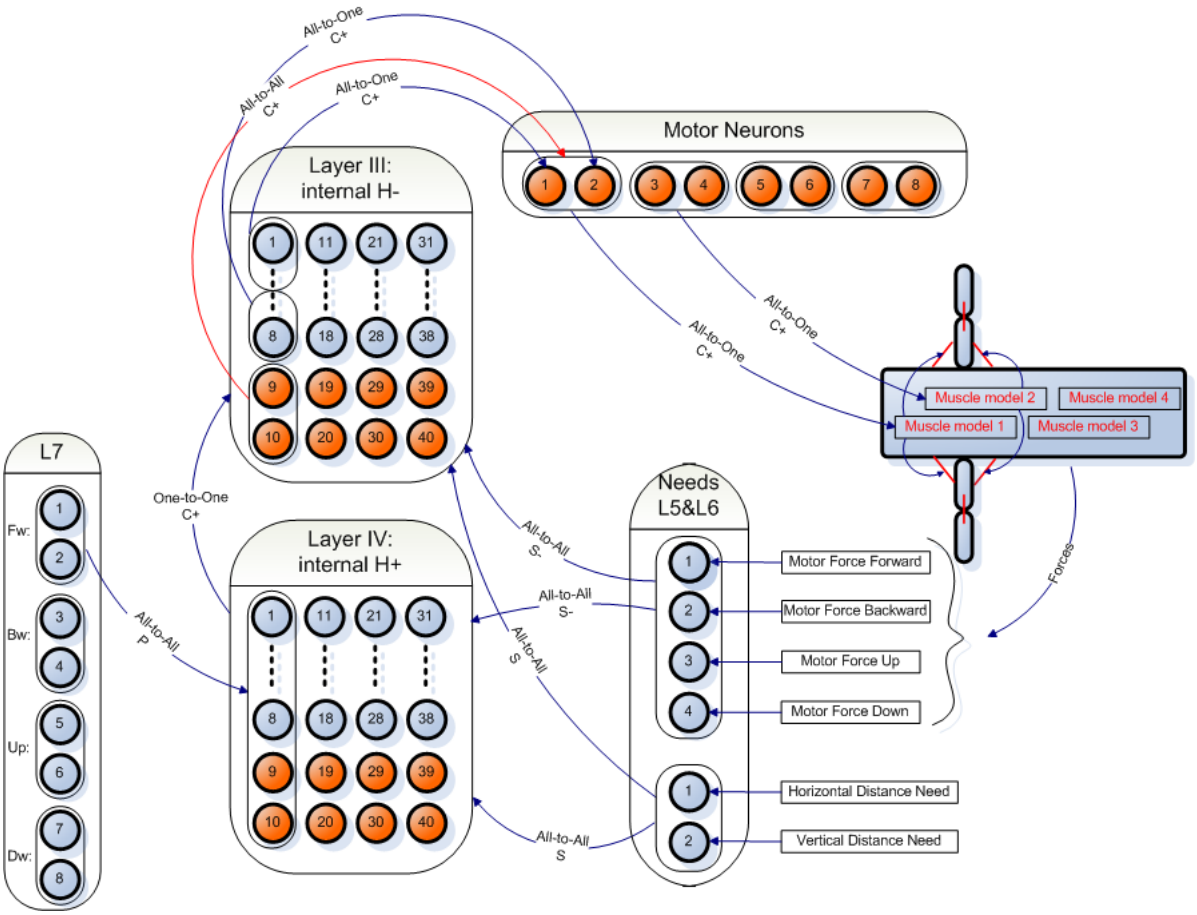

Figure 5.17: The final large topology. The orange neurons represent RS neurons and the blue represent RES neurons. In addition to the synapses seen there are static synapses internally in each subgroup of 10 neurons in layer 3, i.e. between neurons {3.1-3.10} and {3.11-3.20} etc. but there are still H- synapses between neurons in different subgroups. I used S synapses (starting out positive) from layer 6 and S- from layer 5, so that 6 would drive the network towards more activity and layer 5 would seek to bring the activity down in a greater way (not only by lowering the positive weight but actually inhibiting the network activity directly.)

[6]http://www.youtube.com/watch?v=WsUpgw52U7Q&feature=mfu_in_order&list=UL, from this clip we can see that the agent moves a little back and forth which is good for learning, but as we can see at the end it becomes a bit unstable due to focusing too much force on the up muscles.

Parameter	Value	Parameter	Value
SynBeta:	1	A_{Syn} L3	2.0
LearningConstant:	40	A_{Syn} L3->L5	0.005
ForgettingConstant:	1.0E-5	A_{Syn} L3->L6	0.005
STDPLearningConstant:	0.0	A_{Syn} L3->Motor	7.0
WeightUpdateFreq:	100	A_{Syn} L4	0.03
Random burst:	0.0	A_{Syn} L4->L3	3.0
Random weights:	1.0	A_{Syn} L5->L3	0.5
Initial Weights:	-	A_{Syn} L5->L4	1
Internal delays:	5	A_{Syn} L6->L3	0.3
Inter layer delays:	5	A_{Syn} L6->L4	1.0
SpikeRateType:	LargeWindow	A_{Syn} L7->L4	5.0
WindowSkinner{L1,Interval,L2}:	{1600,2100,2600}	A_{Syn} MotorForces->L5	2.0
WindowPavlov{L1,Interval,L2}:	{2600,1800,1600}	A_{Syn} DistanceNeeds->L6	3.0
WindowHume{L1,Interval,L2}:	{2600,1600,1600}	*20% RS neurons in*	*L3&L4 + 100% in the Motor layer*

Table 5.6: The final parameter set for the large topology.

Chapter 6

Discussion, Conclusion and Further Work

This chapter sums up the results of Chapter 5 and discuss why I got the results that I did. Then a conclusion is drawn, and finally I will share my thoughts on what I think should be focused on in future work.

6.1 Discussion

The first thing to notice is that the testing with the mechanical agent demonstrated that the ANN and the muscle model worked as intended together with the Webots application. The movement of the agent's arms were smooth and controlled[1]. The same words can not be used to characterize the movement when I tried to control the agent manually by using the keyboard to add forces to the different muscles.

As we have seen in Chapter 5 the agent struggled to learn anything helpful. Most of the time the agent would either get stuck due to some weights increasing a lot while others dropped to zero, or the agent would hardly move at all. The exception from this is the final result of the medium topology where the agent managed to move a lot back and forth without getting stuck, nevertheless after waiting hopefully for a long time I could still not see any patterns forming. So either the agent was moving around but no patterns formed, or one pattern formed and killed the activity in the rest of the network and the agent got stuck. So I did not manage to obtain any clear alternating patterns. I believe this is practically impossible for the minimal topology because every neuron is too dependent on every other neuron. If one neuron increase its spike rate the neurons connected to it will also do the same right after (in the case of excitatory synapses). In the small topologies I think it is like finding a needle in a haystack. It may be possible but you would have to be lucky or be in possession of a strong magnet, and even then it would not be easy. The medium topology had the most promising results. The only thing that was missing was the desired learning. This could be an indication that the learning rules does not work in practice, or it could be a number of other reasons as I will get back to in the next subsections. The large topology was the hardest to tune because it is way to complex to get any helpful information out of single synapses. I had to look at the whole and use the experience I gained from the smaller topologies, but I could not find any settings that improved the results of the medium topology.

6.1.1 Finding the Balance

I believe finding the balance between the different inputs is one of the major issues. How much influence should the muscle length and velocity neurons have versus the need neurons? If the need neurons have to much influence the internal neurons will rise and fall as the need neurons does, and we will not have much learning. If the length and velocity neurons have to much influence the need neurons will learn, however the learning will

[1] Apart from the few times the spike rate in the network got out of hand.

not have much influence. The balance may also start out good and then later be disrupted by the learning, i.e. if the synapses from the need neuron increase as the agent get closer they may suddenly have a too big influence on the activity of the internal layers so that the internal activity mainly varies in the same pattern as the need neurons.

6.1.2 Different Topologies

All of my topologies are as mentioned based on a particular topology described in Hokland2011[4] which in turn is based on cortical research. However it is not sure that that particular network structure is the easiest way to obtain this kind of self-organization.

Basically there is an endless set of different topologies, and when the intuition does not work as wanted, it could be a good idea to try using an evolutionary algorithm to find a good topology.

6.1.3 The Initial Weights

The way the initial weights are set has a great influence on the network. I started out with small initial weights of ± 0.05, but found that a little randomness often was good for the behavior. I therefor ended up using random initial weights in the range (0, 1). The disadvantage with random weights is of course that each run with the same parameters may produce very different results. But all in all it led to more movement and thereby a better basis for learning. The agent would some times get stuck due to an unlucky weight distribution. However such cases could usually be discovered within a few seconds of simulation. Using random weights leaves more to chance, but the alternative of getting a very homogeneous network is not very appealing either.

6.1.4 The Learning Constants

Another important issue is how to set the value of the learning constants, i.e. how fast should the agent learn, how big effects should each weight update have? Generally I would think that if our time was not an issue, very small changes would give the best result over a long period of time, but since time is valuable we have to compromise to be able to detect any results in a limited period of time. This is essential when there is a great number of parameters to be tuned and tested. It may be that my experiments would be more successful if i lowered the learning constant and let the simulation run over a much longer time period. Sadly I could not prioritize the time that way.

6.1.5 The Time Between Changes

As mentioned in Chapter 3 all of Hokland's learning rules are defined to update the synaptic strength based on events (clear spike rate changes) in the pre- and postsynaptic neurons where one happened before the other. The main challenge here is to figure out how long the interval between the two events should be. If I define this time window wrong I risk that I will miss the right changes and update the weight according to some random changes.

6.1.6 How to Represent the Input

How to represent the input is also crucial. It is really important that the changes in input is reflected in a clear way in the network, i.e. if the agent moves closer to the goal the spike rate in the horizontal distance neuron have to drop. I believe I found a good solution for this, but it could probably be improved a bit, and always has to be adopted to how the original value varies.

Summary

There are generally a lot of variables that have to be tuned right to get the wanted behavior, and a tiny change to one of them might effect the network in a huge way. The network topology is also of great importance.

6.2 Conclusion

I have successfully implemented and tested Hokland's three learning rules. During the testing phase I discovered that the order of the postsynaptic and presynaptic events was not taken into account in his equations. This led to synapses being updated independently of which event came first. I solved this problem by using a new way of calculating the change in spike rate, and modified the learning equations accordingly.

I have made an application consisting of an artificial neural network and a muscle model which can be used to control the muscles of a simulated robot in a smooth and controlled way. However as most others I did not manage to make it self-organize based on Hokland's learning rules. As we have seen this does not necessarily mean that the rules themselves are wrong, since there are many other factors that could also be the problem. Even though none of the topologies and parameter settings I tried resulted in the desired learning, I felt I was close with the final settings of the medium topology. The agent moved quite well back and forth and I was just waiting for a movement pattern to emerge, but that did not happen.

The learning rules works as intended in theory, but in practice I believe there is still some work to be done before we will see any self-organization.

6.3 Further Work

I would start by trying to solve or avoid the challenges described in the discussion section. The way I see it there are three alternative approaches to solve the issues. One could design/use analytical tools to help one study the different issues, as I have done to some degree. Another approach would be to dive deeper into the biological nervous system and hope to find some more answers there.

One could also try to find the right topology and parameter set by using an evolutionary algorithm (EA). A challenge with this approach is to find a good way of estimating the success (fitness) of each topology and parameter set. In my case an initial thought would be to use the inverse distance to the goal as the fitness function, but it takes a lot of time to run these 3D simulations and an EA demands a lot of runs to accomplish its purpose [2]. Therefor I would think that one would have to find a faster way to evaluate the fitness, i.e. by dropping the visualization of the mechanical model and just evaluate the pattern of the forces generated by the muscle model. And then hope that one took into consideration all the essential parts necessary for it to function in a real environment.

One could probably avoid some of the challenges by redefining the experiment i.e. by using a different or no physical model, using other inputs or other needs to name some.

It is an exciting field to study, and I believe that we will see this kind of learning some time in the future. When this is going to happen is impossible to predict, but as computers get faster we will be able to run more and different test in a shorter period of time which will open a lot of new doors.

One thing I think would be interesting to try out, is a new way of setting up the network. The way I imagine this is to define large neuron groups (or super neurons as I would call them), consisting of about 500 neurons each, and let the spike rate of a neuron group be defined as the number of internal neurons spiking at any given time. This way the spike rate would be very local in time but also stable, and it would not take much resources to calculate. Figure 6.1 shows a spike rate calculated this way. The weights between the neurons in one group would be static to maintain a robust "super neuron" and lower the run time considerably. In each neuron group there would be a mixture of RS and RES neurons[3]. Each super neuron would then be connected to other super neurons by different types of synapses. Figure 6.2 shows an example of how I imagine it. The advantage of this would be a robust, stable and responsive network, with a local but good spike rate estimate. The disadvantage would be the run time due to the amount of neurons to update.

[2] Basically EAs work by starting out with some different parameter sets. These are evaluated by a fitness function, and the best parameter sets will be allowed to have some children (i.e. by mixing two and two of the best parameter sets), while the bad parameter sets "die". There will then be some random changes (mutations) to some random parameters, before the new parameter sets will be evaluated and the best will survive and have children, and so on.

[3] In the project leading up to this project I found that a distribution of 80% RES neurons (where 20% of them was inhibitory neurons) and 20% RS neurons worked very well.

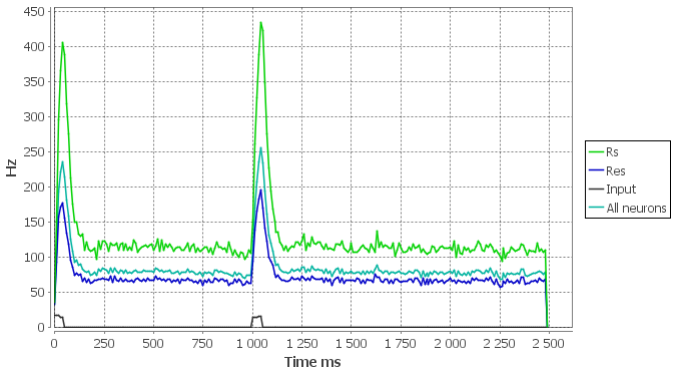

Figure 6.1: Shows the spike rate of the RES neurons (blue), the RS neurons (green), and the average spike rate (teal) taking into consideration that there are 80% RES neurons and 20% RS neurons. The input to the network (super neuron) is shown in black. The average spike rate would determine the output of the neuron group. In this case the spike rate estimate is calculated by averaging over the last ten time steps.

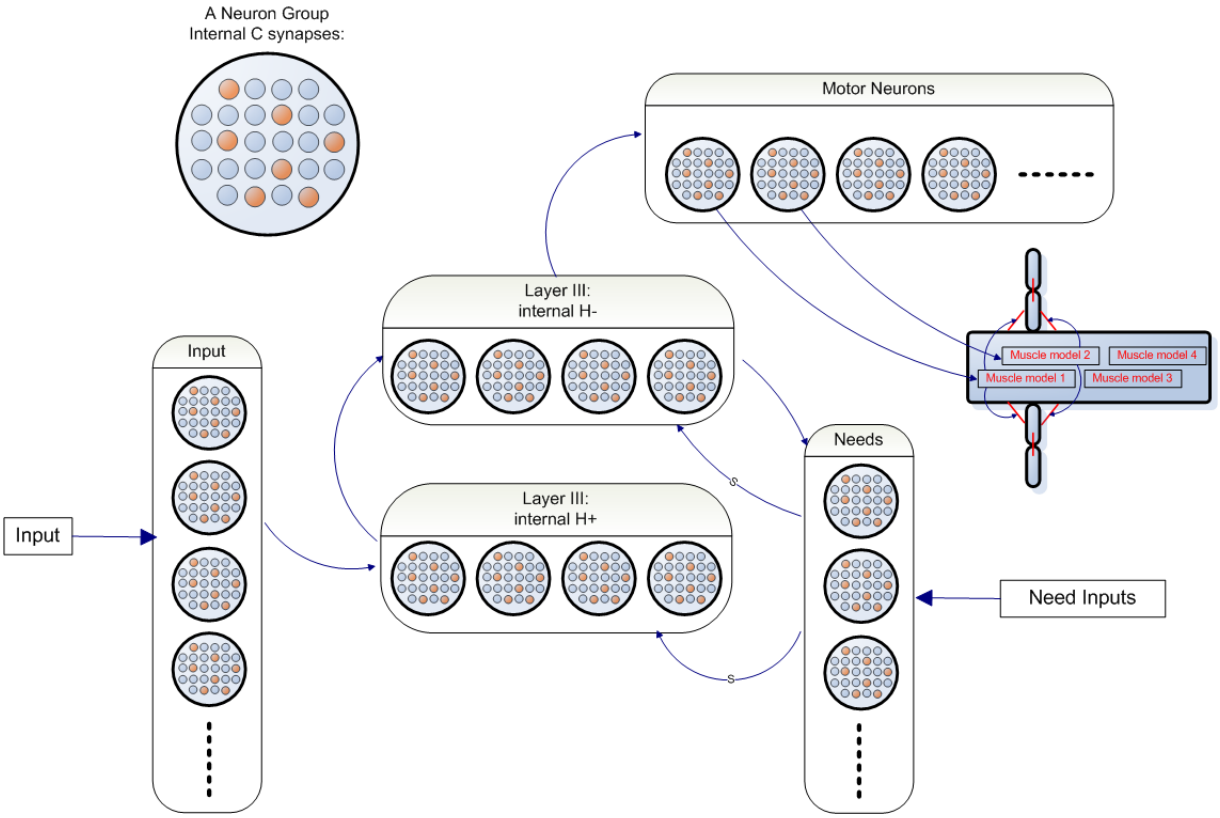

Figure 6.2: A topology of "super neurons". Basically exchanging the regular neurons with large neuron groups.

Appendix A

The Application Parameters

A.1 The GUI

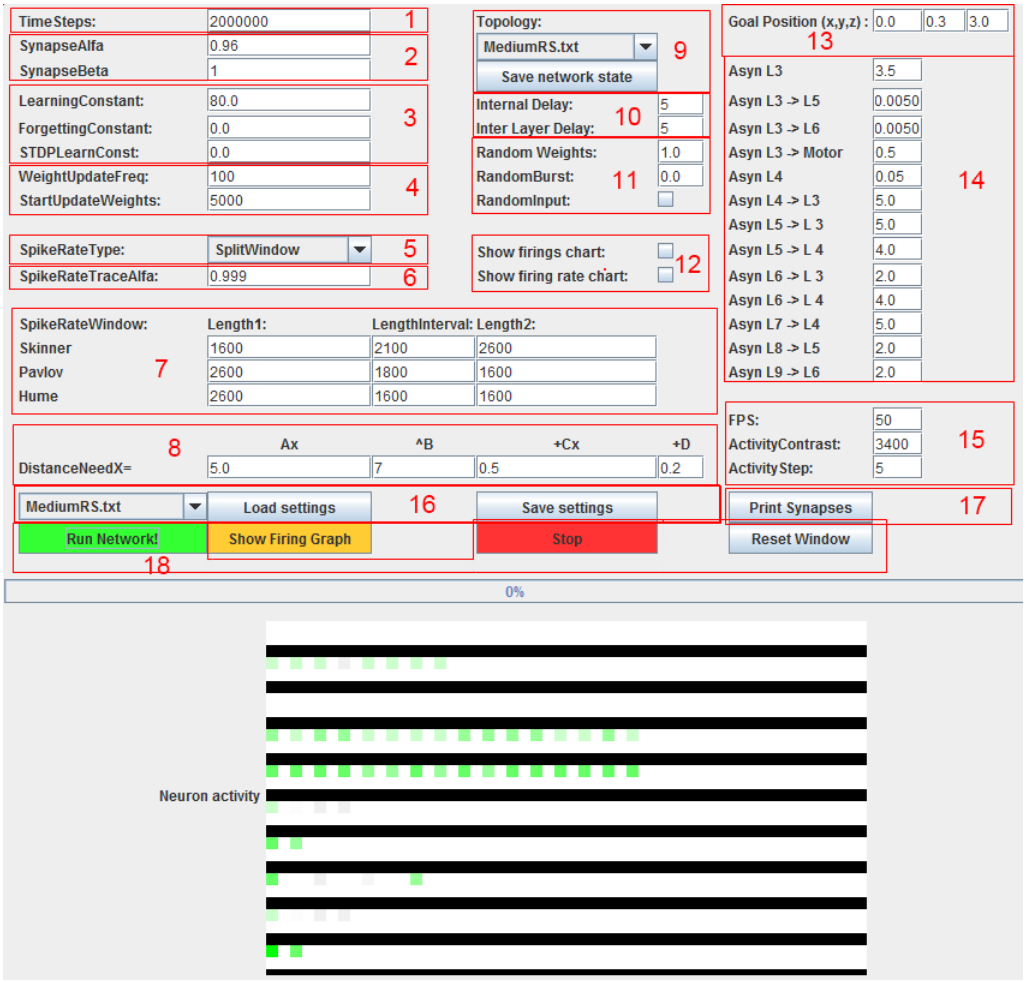

Figure A.1: The GUI.

1) TimeSteps

The length of the simulation, when used in the robot simulation the length is set to 'infinity'.

2) SynapseAlpha and SynapseBeta ($\alpha_{\mathbf{Syn}}$ and $\beta_{\mathbf{Syn}}$)

This is the values used in Hokland's learning rule formulas , see Equations 3.1 to 3.4 in Section 3.1.

3) LearningConstant, ForgettingConstant and STDPLearningConstant (C_{Hok}, C_{Forget} and C_{STDP})

The first two (C_{Hok} and C_{Forget}) are the learning constants used in Skinner, Pavlov and Hume synapses as described in Equation 3.5 to Equation 3.10. While the STDP constant is the one used in Equation 3.11.

4) WeightUpdateFreq and StartUpdateWeights

WeightUpdateFrequency is an integer that defines the interval between each weight update.

StartUpdateWeights specifies the time of the first weight update, this is used so that the initial increase from zero to some spike rate will not effect the weight.

As mentioned in Section 3.1 a weight update frequency of 1 will not lead to the desired weight updates as we can see in Figure A.2.

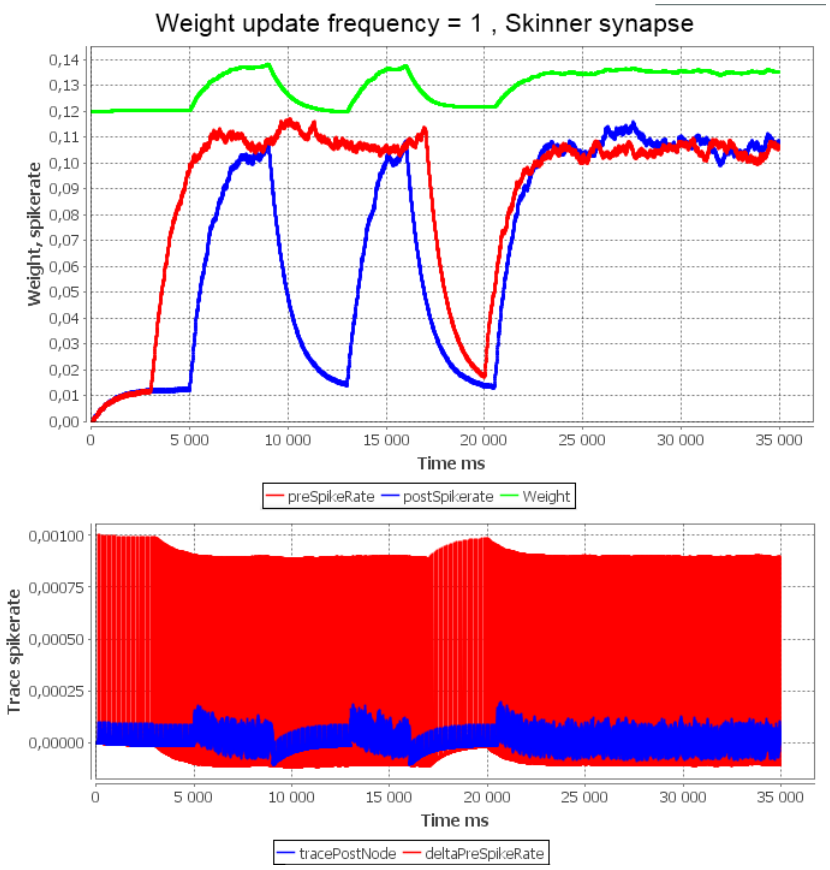

Figure A.2: Using a weight update frequency of one.

5) SpikeRateType

This is a drop down list from where one can select one of the four spike rate estimation methods described in Chapter 4.

6) SpikeRateTraceAlpha ($\alpha_{\mathbf{Trace}}$)

This is the value that decides the length and amplitude of the trace, used in Equation 4.1.

7) SpikeRateWindow Parameters

This is a table that describes the window length for the three different synapse types, the parameters are described in Section 4.4.2. What is important to notice is that Length1 and Length2 have to be even integers since the windows are split in two when calculating spike rates.

8) DistanceNeedX

Defines the different parameters of the need amplifier equation in Section 2.1.2. It is used to get clear drops in the spike rate when the agent is moving closer to the goal. (As mentioned earlier similar equations are used for the other needs and muscle inputs but these can not be set from the GUI.)

9) Topology

This is a drop down menu where one selects the desired network topology. The network topology is defined by certain key-words in a .txt file which later is parsed by the application and used to generate all the layers, neurons and synapses. The current state of the network (i.e. the layers, neurons and the synapses with their weights) can also be saved to a .dat file by hitting the "Save network state" button. This file would also appear in the drop down list and can be selected as an alternative to the text file.

10) Delays

These two fields can be used to specify the general delays of synapses internally and between layers. The effect of the delays is that if the presynaptic neuron spikes and there is a delay of 5, it would take 5 time steps before the postsynaptic neuron registered this spike. If the value of these fields are set to -1 the delay will be as specified in the topology file.

11) RandomWeights, RandomBurst, RandomInput

If the random weights value X is greater than zero, the weights of the synapses in the network will initially be set to a random value in the range (0, X).

The random burst value Y will lead to a second of increased activity in a neuron every time a random value in the range (0, 1) is less than Y, regardless of the membrane potential v.

The random input check box is used when running the ANN without the Webots part of the system. This is used for the initial testing of the implementation of the learning rules and some pre-tuning of the different parameters.

12) Show Charts

Select whether or not to display a plot of all the firings of every neuron, and the spike rate charts. The firings chart can also be displayed at any time by clicking the orange button

13) Goal Position (x,y,z)

Specifies the coordinates of the food source.

14) Asyn (A_{Syn})

This is one of the factors determining the strength of a synapse, the value can be defined differently in and between each of the layers. Unlike the weight, the A_{Syn} value is constant throughout the entire simulation. The value is used in Equation 1.5.

15) GUI display parameters

These three parameters are used to set the update frequency and the contrast of the display below the progress bar.

16) Settings

All the parameters specified in the GUI can be saved to a text file which will appear in the drop down list. The different setting files can later easily be loaded by selecting them in the drop down list and hitting the "Load settings" button.

17) Print Synapses

This button can at any time print all the synapses of the network, including the presynaptic and postsynaptic neuron and the current weight.

18) Run Network, Stop, Reset Window

Starts and stops the simulation. And resets the window.

A.2 Example Code

This section shows a couple of code examples, the entire source code is attached in a zip file along with the report.

A.2.1 Topology File Example

```
topology.txt

##### START OF NEURAL NET TOPOLOGY FILE #####

# SYNTAX
#layer layerId neuronCount neuronType name
#syn from_layerID from_neuronID synapse_type to_layerID to_neuronID delay weight
#synall fromLayerID synapse type toLayerID delay weight
#syn_1_to_many fromLayerID synapse_type toLayerID seq_mod neuronsPrRow delay weight
#syn_1_to_1 fromLayerID synapse_type toLayerID delay weight
# # -> comment

##### THE LAYERS #####
#Syntax: layer layerId neuronCount neuronType name

# Layer 7 Input layer, muscle length and muscle velocity
layer 7 8 SENSOR Inp_MusAcc

# Layer 4, Internal layer
```

```
layer 4 16 RES Layer_IV

# Layer 3, Internal layer
layer 3 16 RES  Layer_III

# Layer 8, Transforming Golgi output to input for layer 5
layer 8 4 SENSOR Inp_Golgi

# Layer 9, Distance Needs input layer
layer 9 2 SENSOR Inp_Distance_Needs

# Layer 5 Motor force need layer, receive input from layer 8
layer 5 4 RES Layer_V

# Layer 6 Distance need layer, receive input from layer 9
layer 6 2 RES Layer_VI

# Layer 1 Motor Neurons input from layer 3
layer 1 8 MOTOR Out_Motor


##### SYNAPSES #####

# Layer 3
synall 3 H- 3 1 -0.05 skipRow 4
#creates H- synapses from layer 3 to layer 3, with an initial weight of -0.05,
#but it skips the  H- synapses between neurons in each sub group

synall 3 P+ 5 1 0.05
synall 3 P+ 6 1 0.05

#3->motor
syn 3 1 ST 1 1 1 1
syn 3 2 ST 1 1 1 1
syn 3 3 ST 1 2 1 0.8
syn 3 4 ST 1 2 1 0.8
syn 3 5 ST 1 3 1 1
syn 3 6 ST 1 3 1 1
syn 3 7 ST 1 4 1 0.8
syn 3 8 ST 1 4 1 0.8
syn 3 9 ST 1 5 1 1
syn 3 10 ST 1 5 1 1
syn 3 11 ST 1 6 1 0.8
syn 3 12 ST 1 6 1 0.8
syn 3 13 ST 1 7 1 1
syn 3 14 ST 1 7 1 1
syn 3 15 ST 1 8 1 0.8
syn 3 16 ST 1 8 1 0.8

# Layer 4
synall 4 H+ 4 1 0.05
syn_1to1 4 ST 3 1 1
```

```
# Layer 5
synall 5 S- 3 1 -0.05
syn 5 1 S- 4 1 1 -0.05
syn 5 1 S- 4 2 1 -0.05
syn 5 1 S- 4 3 1 -0.05
syn 5 1 S- 4 4 1 -0.05
syn 5 2 S- 4 5 1 -0.05
syn 5 2 S- 4 6 1 -0.05
syn 5 2 S- 4 7 1 -0.05
syn 5 2 S- 4 8 1 -0.05
syn 5 3 S- 4 9 1 -0.05
syn 5 3 S- 4 10 1 -0.05
syn 5 3 S- 4 11 1 -0.05
syn 5 3 S- 4 12 1 -0.05
syn 5 4 S- 4 13 1 -0.05
syn 5 4 S- 4 14 1 -0.05
syn 5 4 S- 4 15 1 -0.05
syn 5 4 S- 4 16 1 -0.05

# Layer 6
synall 6 S+ 3 1 0.05
synall 6 S 4 1 0.05

# Layer 7 - Muscle length and velocity(positive)
syn_1_to_many 7 P+ 4 seq 4 2 1 0.05
# groups two and two neurons in layer 7
# to 4 neurons in sequence in layer 4
# i.e. 7.0&7.1->4.0-4.3


# Layer 8 - Golgi needs
syn_1to1 8 ST 5 1 1

# Layer 9 - Other need drives
syn_1to1 9 ST 6 1 1

##### END OF NEURAL NET TOPOLOGY FILE #####
```

A.2.2 The Main Loop of the ANN

```
while(currentTime<timeSteps){
    // Update all nodes in all layers:
    for(int i=0;i<layers.length;i++){
        if(layers[i] != null){
            layers[i].updateNodes();
        }
    }

    // Update all synapses:
    for(int i=0;i<synapses.length;i++){
        synapses[i].updateSpikes();
```

```
        }

        // Update weights:
        if((currentTime)%z.getWeightUpdateRate() == 0) {
            for(int i=0;i<synapses.length;i++){
                synapses[i].updateWeight();
            }
        }

        //update display
        if(currentTime%(int)(1000/z.getDefaultFPS()) == 0) {
            display.update();
        }

        currentTime++;

        //update the progress bar
        gui.getProgressBar().setValue(currentTime);
    }
```

A.3 Skinner and Hume "Forgetting Constant"

Figure A.3 show the extended Skinner and Hume rules with $C_{Forget} = 1.0E - 5$.

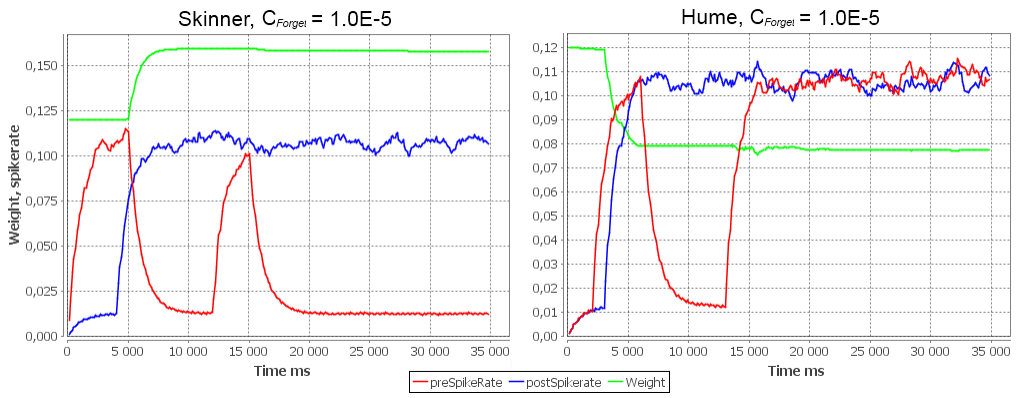

Figure A.3: The extended Skinner and Hume rules. The graphs first show a normal weight update, and then from 15 000 ms (Skinner) and 13 000 ms (Hume) we can see how the synapse forgets a little.

Bibliography

[1] Vebjørn Wæ rsted Axelsen. *Self-Organized Synaptic Learning of Gaits in Virtual Creatures*. Master's thesis, NTNU, 2007.

[2] Jø rn Hokland. A Hebbian Neural Model of Classical and Instrumental Conditioning. 1997.

[3] Jø rn Hokland. *Principia Psychologica*. 2010.

[4] Jø rn Hokland. On the Meaning of Synapses: Skinner, Pavlov, & Hume Rule. 2011.

[5] E M Izhikevich. Simple model of spiking neurons. *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 14(6):1569–72, January 2003.

[6] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 15(5):1063–70, September 2004.

[7] Eugene M Izhikevich. Polychronization: computation with spikes. *Neural computation*, 18(2):245–82, February 2006.

[8] Eric Kandel, James Schwartz, and Thomas Jessell. *Principles of Neural Science*. McGraw-Hill Medical, 2000.

[9] Raul C Muresan and Cristina Savin. Resonance or integration? Self-sustained dynamics and excitability of neural microcircuits. *Journal of neurophysiology*, 97(3):1911–30, March 2007.

[10] Dale Purves. *Neuroscience, Fourth Edition*. Sinauer Associates, Inc., 2007.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.