STOCKHOLMS
UNIVERSITET

Towards a Biologically Plausible and Efficient Reinforcement Learning Agent

Thomas Proschinger

TRITA-NA-E05161

Numerisk analys och datalogi
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Towards a Biologically Plausible and Efficient Reinforcement Learning Agent

Thomas Proschinger

TRITA-NA-E05161

Master's Thesis in Computer Science (20 credits)
Single Subject Courses,
Stockholm University 2005
Supervisor at Nada was Christopher Johansson
Examiner was Anders Lansner

Towards a Biologically Plausible and Efficient Reinforcement Learning Agent

Abstract

The primary goal of this degree project was to evaluate a selection of biologically plausible reinforcement learning (RL) agents. An agent was assumed to be biologically plausible if it could be constructed using a biologically plausible model of learning. Such a model had to be realizable using an artificial neural network (ANN) with a learning rule that could be implemented within biological constraints. The first step of this project consisted of reviewing the literature on this type of model. Four different models were found. Two of the models were used to construct RL agents: the S-model A_{R-P} weight update rule, and the dual projection BCPNN RL system.

The agents were experimentally evaluated by presenting them with a set of challenging tasks. Performance measurements were gathered as they solved the tasks. An implementation of the Sarsa algorithm was used for benchmarking. The evaluations illuminated different strengths and weaknesses of the two agents. Neither agent was found to be very efficient. However, the insights gained could still be used to construct more efficient biologically plausible RL agents in the future. Other insights gained from this project include: (i) a simple and general framework for constructing RL agents from biologically plausible models of learning, and (ii) an evaluation procedure which proved to be very useful in illuminating different features of the agents.

Mot en biologiskt plausibel och effektiv agent för förstärkningsinlärning

Referat

Det primära målet med detta examensarbete var att utvärdera ett urval av biologiskt plausibla agenter för förstärkningsinlärning (RL). En agent antogs vara biologiskt plausibel om den kunde konstrueras utifrån en biologiskt plausibel inlärningsmodell. En sådan modell skulle vara realiserbar med ett artificiellt neuralt nätverk (ANN) vars inlärningsregel kunde implementeras under biologiska inskränkningar. Första steget i exjobbet gick ut på att söka i litteraturen efter den här typen av modell. Fyra olika modeller hittades. Två av modellerna användes för att bygga RL agenter: S-modellens A_{R-P}-viktuppdateringsregel, och det dubbelprojicerande BCPNN RL systemet.

Agenterna utvärderades försöksvis genom att låta dem lösa ett antal uppgifter. Data om deras prestation samlades in medan de löste uppgifterna. En implementation av Sarsa-algoritmen användes som måttstock. Experimenten belyste olika styrkor och svagheter hos de två agenterna. Ingen av agenterna ansågs vara särskilt effektiv. Insikterna som uppnåtts kan dock ändå användas för att konstruera effektivare biologiskt plausibla RL agenter i framtiden. Andra insikter som detta projekt har gett upphov till inkluderar: (i) ett enkelt och generellt ramverk för att konstruera RL agenter utifrån biologiskt plausibla inlärningsmodeller, och (ii) en utvärderingsprocedur som visade sig vara mycket användbar för att belysa olika egenskaper hos agenterna.

Foreword and Acknowledgments

This thesis presents my Master's project in Computer Science. It was carried out within the research group of Computational Biology & Neurocomputing (CBN) at the Department of Numerical Analysis and Computer Science (Nada), Stockholm University. The project examiner was Professor Anders Lansner, whom I would like to thank for his ideas and advice. Christopher Johansson, my supervisor at Nada, deserves my gratitude for all his support and valuable feedback throughout this project. Instructor Caroline Nordquist at Nada did a great job reviewing and improving the quality of this thesis, thank you. Last, but not least, I want to express my gratitude to my family. Thank you for always being there for me.

Thomas Proschinger, Stockholm 2005

Contents

Chapter 1

Introduction

From an engineer's point of view, the human brain is an amazing piece of machinery. For one, the brain allows us to control our bodies to do our will, whether it involves walking, running, swimming, typing, climbing, or picking up an egg without crushing it. However, body control is only one requirement to get these tasks done. Another one, just as important, is our ability to perceive our surroundings by signals passed to us via our five senses. These signals are processed by the brain almost instantly, and allows us to understand and react to our environment appropriately. The adaptivity of the brain also allows us to master an incredibly diverse set of skills, such as reading, writing, driving, medical diagnosis, teaching, diving, and so on. All these skills are readily available to us, should we wish to learn them.

The superiority of the human brain becomes obvious whenever we attempt to create so called "intelligent" software. By this we mean that the software, in some respect, behaves in an intelligent way. Trying to program software capable of doing any of the things mentioned above is daunting. Even if we sometimes, in some limited field, can achieve a reasonable amount of success, the brain is usually faster and more efficient.

It seems only natural to look to the brain for inspiration in building intelligent and adaptive software. Unfortunately, the inner workings of the brain is to a large extent shrouded in mystery. We do know that the brain processes information in a fundamentally different way than a computer does. A conventional computer typically has a single powerful processor. The human brain, on the other hand, contains a staggering number of simple and massively interconnected processing elements known as neurons. The neurons in our brain connect with each other at sites called synapses to form a single complex network. There are approximately 100 billion neurons in the human brain, and the number of synapses is about 1000 trillion (Tortora and Grabowski, 2003, p. 452). The knowledge of the brain is believed to be stored at these synapses, see Section 2.2 on page 10. This knowledge is continuously updated throughout life, because the brain never really stops adapting and learning. A remarkable feature, which traditional software completely lacks.

1.1 Previous Work

Artificial Neural Networks (ANNs) are designed to model the way we think the brain processes information. ANNs have turned out to be powerful computing tools. Their virtues include the ability to compute any computable function, and they are also noise

and fault tolerant, as is the brain (Gurney, 1997, pp. 16, 17; Haykin, 1999, pp. 2–4). ANNs have been used for a wide variety of applications over the years. Examples include: medical diagnosis, handwritten character recognition, voice recognition, and stock market prediction. Also, because they are designed to mimic the real thing, these efforts have led to a deeper understanding of how the brain itself might solve these tasks. The knowledge of an ANN is stored in its so called weights. A weight is the artificial equivalent to the strength of a synapse, see Section 2.2 on page 10. Learning, the acquisition of knowledge, is accomplished by updating those weights. The set of algorithms which do this are collectively referred to as the ANN's weight update rule. Researchers in this field are usually more concerned with efficiency, rather than trying to create biologically plausible weight update rules. This is not to say that such rules do not exist, but they are unpopular. As a result, most of the ANN literature do not provide us with many clues as to how how the brain might learn.

Reinforcement learning (RL) is a learning paradigm within the field of machine learning. RL provides a framework on the problem of learning by interaction to achieve a goal. Put simply, its about learning by doing and observing what happens. It is a very intuitive approach to learning, as it is quite similar to how we humans can learn by the method of trial and error. This goal-directed learning is something we know the brain can do very well, but it is currently poorly understood how. A contributing factor is no doubt due to the fact that most of the previous work in this field has been completely unrelated to biology. The focus has rather been on the development of efficient algorithms. In that respect, much progress has been made. However, because these algorithms have been developed without any biological considerations, there is nothing in them that can tell us anything about how the brain solves RL problems, i.e. how it learns by interacting with its environment. This is something we would very much like to know, since while several successful RL algorithms have been developed, they are not even remotely able to compete with the brain on real-world problems. The reason for this is partly due to poor scaling characteristics.

ANNs have been used in the field of RL before, but only in relatively few cases has the RL agent been directly implemented with ANNs. For the most part, ANNs have been used merely as supportive tools, often in the form of function approximators. One of the most famous and successful applications of an ANN as a supportive tool in the context of RL was TD Gammon, by Tesauro (1992, 1995). His application was able to learn to play the game of backgammon at an expert level, with no game specific knowledge built in. It achieved this by playing many thousands of games against itself. At the core of the application was an ANN, which was used as a function approximator to predict the outcome of game positions. Specifically, the ANN predicted the probability of either player winning a normal win, or a so called gammon. The network was trained by the methods of temporal difference (TD) (Sutton, 1988). A simple look-ahead mechanism was used to select moves based on the predictions of the ANN.

1.2 Goal and Purpose

The primary goal of this project was to evaluate a selection of biologically plausible RL agents. The purpose of doing this was primarily to advance our knowledge of how such agents can be constructed and evaluated, but also to gain insights into how to potentially make them more efficient. Ultimately, hopefully within a not too distant future, we will be able to construct a biologically plausible RL agent that is efficient enough to rival the human brain on solving real-world problems. Furthermore, the

process of trying to develop such an agent could potentially help us gain insights into how the brain itself accomplishes goal-directed learning.

An RL agent was defined to be biologically plausible if it could be implemented using a biologically plausible model of learning. Such a model had to be possible to realize using an ANN with a biologically plausible weight update rule (see Section 4.2 on page 22). It was important to show how the construction of the agent, using such a model, was carried out. Development of biologically plausible RL agents has not seen much research in the past. Only one such agent was known to have been implemented previously.

A cornerstone of this project was the assumption of what makes a model of learning biologically plausible. To motivate the selection of models, it was important to also review what literature there is connecting RL and ANNs with the biology of learning in real neural networks. Of course, finding these models in the first place was a priority.

1.3 Restrictions

Many interesting biologically plausible models of learning were found, especially from the field of classical conditioning, which could potentially be used to build biologically plausible RL agents. Classical conditioning is a particularly rich source, because learning in those models is driven by stimulus signals; much like how RL agents learn from the reward signal. Furthermore, these models often have a neural substrate, or are based on a neural substrate. It seems reasonable to assume that many models from that field could be used to build biologically plausible RL agents. Due to time restrictions, only four of the models found will be described in this thesis. Of those, only two will be evaluated.

Interesting sources for more models, none of which are described in this thesis, include: Balkenius and Morén (1998), Bartlett and Baxter (1999), and Wörgötter and Porr (2005).

1.4 Thesis Structure

The structure of the thesis is as follows.

Chapter 2 Provides the theoretical background of the project. The topics covered are: RL, some basic neurobiology, and ANNs.

Chapter 3 Describes the four biologically plausible models of learning that were found in the literature.

Chapter 4 Explains the method of evaluation. The chapter includes a section which motivates the selection of the models. It also describes how the RL agents were constructed from the models. Finally, the chapter details the specifics of the evaluation procedure.

Chapter 5 Presents the results of the evaluations.

Chapter 6 Discusses the evaluation results. Conclusions are drawn, and suggestions are made for future work in this area of research.

1.5 Abbreviations and Symbols

Here follows a summary of important abbreviations and symbols used in this thesis.

Table 1.1 Important abbreviations and symbols.

Abbreviation	Explanation
2-AB	2-Armed Bandit
ANN	Artificial Neural Network
AP	Action Potential
BCPNN	Bayesian Confidence Propagating Neural Network
LTP	Long-Term Potentiation
PSP	Postsynaptic Potential
RL	Reinforcement Learning
TD	Temporal-Difference
Symbol	**Definition**
$\mathcal{A}$	set of all actions available to the agent
E	number of learning episodes used in every run R
R	number of runs used to evaluate the agent
R_t	return at time step t, the (discounted) accumulated reward received from time step $t+1$ onward
$\mathcal{S}$	set of all non-terminal states
$\mathcal{S}^+$	set of all states the environment can occupy
a_t	action chosen by agent at time step t
h_i	support value of an agent for action i
m	number of actions available to the agent, $m = \|\mathcal{A}\|$
n	number of non-terminal states, $n = \|\mathcal{S}\|$
p_i	probability of choosing action i
π_t	policy at time step t, maps each non-terminal state to a probability distribution over the actions available in that state
r_t	reward received by the agent at time step t
s_t	state of the environment at time step t
u_{ij}	episode value, the value of episode j in run i
u_i	run value, the value of run i
$u_{\max}$	best-case performance, taken over all runs R, of an agent on a specified task
$u_{\min}$	same as $u_{\max}$, except this is the worst-case performance
w_{ij}	weight which modulates the connection from input node or unit j to unit i
x_j	input signal from input node j which connects to one or more units

Chapter 2

Background

This chapter provides the theoretical background of this project. The first section covers the essentials of RL. After that, some basic neurobiology is introduced, and some connections back to the field RL are made. The section on neurobiology hopefully makes the last section about ANNs intuitive to grasp.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a subfield of machine learning: the study of algorithms/systems that can improve their performance through experience. The learning system typically accumulates this experience through an iterative process. During the iterative process, input patterns are presented to the system one at a time. If the system does not get any more information than the input patterns, the system is said to undergo unsupervised learning. This is to be contrasted to supervised learning. That kind of learning provides the system with information about the correct response to each input pattern, typically in the form of an error signal. The error signal carries information about how "off" the system's response was from the correct one. A well-known example of a supervised learning algorithm is the error back-propagation algorithm, see Section 2.3 on page 13. See Haykin (1999, p. 63) for more information about supervised learning.

RL is similar to supervised learning, except its error signal is much simpler. RL is a fairly intuitive approach to machine learning, as it resembles the way we humans can learn by the method of trial and error. In trial and error, one tries an option to see if it is satisfactory. If it is, we have accomplished our goal and found a solution. If not, there is an error, and another option is tried.

Formally, RL provides a framework on the problem of learning by interaction to achieve a goal. The learning system, or learner, is called an agent. The idea is to have the agent learn by trying different actions and observing the outcome of these actions. The outcome is provided by the environment, which is what the agent interacts with. See Figure 2.1 on the following page. In response to an action, the agent will receive a numerical value, a reward, which indicates how good or bad its chosen action was. Somewhat informally, the goal of the system is to learn how to maximize the reward it receives. If it has done that, then it has also learned how to achieve the goal. If maximizing the rewards corresponds to achieving the goal, then the problem has successfully been framed as a RL problem.

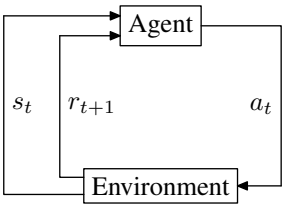


Figure 2.1 The agent-environment interaction.

The agent and the environment interact via three signals: the state s_t, action a_t, and reward r_{t+1}. Each state s_t must belong to the set of possible states, denoted $\mathcal{S}^+$, and every action a_t must be chosen from $\mathcal{A}(s_t)$, the set of all actions available in state s_t. This choice of action is made by implementing a policy π_t, which is a mapping from each state to a probability distribution over the actions available in that state. The rewards are always real numbers.

The interaction begins with the agent receiving state s_0, which specifies the initial state of the environment. The agent responds with an action a_0. As a consequence of that action the agent receives a numerical reward r_1, followed by the next state s_1. Now, unless s_1 is a terminal state (see below), the agent will respond with action a_1, and so the cycle repeats itself.

A complete specification of an environment is called a task. Tasks can be episodic or continuous. They are called episodic if they have a natural ending, such as checkmate in a game of chess. This "end state" is formally called a terminal state. Continuous tasks, on the other hand, do not have such a natural ending. While the set of all states is denoted $\mathcal{S}^+$, the set of all non-terminal states is denoted $\mathcal{S}$. Usually only $\mathcal{S}$ is on interest to the agent, since no action is to be taken in a terminal state.

An episode[1] is a sequence of states, actions, and rewards, beginning with state s_0 at time $t = 0$ and ending some finite time $T > 0$ later in a terminal state s_T:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T. \tag{2.1}$$

It was previously stated, informally, that the goal of the agent is to maximize the accumulated reward it receives. Formally, the agent should maximize the expected return. For episodic tasks, the return can be defined as

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T. \tag{2.2}$$

Assuming the rewards are finite, the sum in (2.2) will also be finite since the task is assumed to be episodic. The return is defined slightly differently for continuous tasks, in order to ensure the return only assumes finite values. This is similar to the concept of discounting, which will be introduced next. Discounting can be used to represent that immediate rewards may be worth more than rewards which lie in the distant future. The discounted return can be used for this purpose. It is defined as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-1} r_T, \tag{2.3}$$

where $\gamma \in [0, 1]$ is the discount parameter. In other words, a big reward in the distant future may be less desirable than a small reward right now. Note that (2.3) reduces to (2.2) for $\gamma = 1$. For continuous tasks there is no final time step T, so by setting $\gamma < 1$ it can be ensured that the discounted return always assumes a finite value.

[1] Episodes are sometimes called trials or epochs.

The Exploration-Exploitation Dilemma

This definition of the goal, to maximize the expected return, presents the agent with a problem. In order to maximize the expected return the agent must choose the actions it has found to be favorable. The problem is that in order to discover those favorable actions in the first place, the agent has to explore its environment by trying actions it has not tried before. The agent cannot leave an action untried, as it may be the very action which yields the highest reward. As a further complication, there is no guarantee that the environment is deterministic in handing out rewards. An action may for example yield a reward 90 % of the time. If an agent were to select that action and receive no reward, it may, erroneously, conclude that the action is bad and never try it again. The only way for the agent to discover how good any action really is, is to try it over and over. Of course, the agent may not exclusively explore actions in this manner, since it then would fail to reach its goal: to maximize the expected return. The agent is required to continuously explore its environment while progressively exploit its knowledge of it. The matter is further complicated if the environment is non-stationary, i.e. if it changes over time. However, all tasks considered in this project were stationary. For more information about the exploration-exploitation dilemma, see Sutton and Barto (1998, pp. 26–27, 30).

Value Functions

Value functions are functions that, given a policy π, assign a numerical value to either a state: $V^\pi(s)$, or a state-action pair: $Q^\pi(s, a)$. Their purpose is to provide a numerical estimate of "how good" it is to either be in a state s, or to choose an action a in a state s. A natural measurement for this is the expected return, which is why they are defined as

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} \quad \text{and} \quad Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\}, \tag{2.4}$$

where the $E_\pi\{\}$ expression denotes the expected return received, given that the agent follows policy π from time step $t + 1$ onwards. The return R_t could be defined as in (2.3) or (2.2).

These value functions are typically not known beforehand. Instead, they are estimated based on the agent's experience of interacting with the environment. The estimated value functions are denoted $\hat{V}^\pi$ and $\hat{Q}^\pi$ respectively. To simplify the discussion, only the state-value function $\hat{V}^\pi$ will be addressed from now on. The reasoning will be similar for the state-action value function. Also, the estimated state-value function will be written as simply $\hat{V}$ from now on.

When a particular policy is in place, the agent can use that to explore the environment, thereby obtaining a sample of the return. This sample return can be used to improve the estimated value function by making it more consistent with the policy. A simple example of how this update might be accomplished is

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha(R_t - \hat{V}(s_t)), \tag{2.5}$$

where R_t is the accumulated reward the agent actually received after time step t, and $\alpha \in [0, 1]$ is a constant learning rate parameter. This update rule strives to make $\hat{V}(s_t)$ more similar to R_t. The methods of learning that wait until a sample of the return is known are called Monte-Carlo methods.

Generalized Policy Iteration

The improved estimated value function computed in (2.5) can be used to improve the policy. This is accomplished by making the new policy "greedy" with respect to the current state-value function $\hat{V}$. Making the policy greedy means having it assign higher probabilities to those actions which yield higher values according to the current $\hat{V}$. Next, the current estimated value function may again be improved, in order to match the improved policy better. This is an iterative process, and Sutton and Barto (1998) used the term "generalized policy iteration" (GPI) to refer to the general idea of improving both the policy and the estimated value function in this way.

Temporal-Difference Learning

Sutton and Barto (1998) identified temporal-difference (TD) learning as a central idea in the field of RL. Unlike Monte-Carlo methods, TD methods do not have to wait for a sample return to become available in order to make a useful update of $\hat{V}$. The basic idea of TD methods is that learning is based on the difference between temporally successive estimates (Tesauro, 1995). The most basic TD method, known as TD(0), has the following update rule:

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha(r_{t+1} + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)), \tag{2.6}$$

which makes use of the discount rate parameter in (2.3). It is quite intuitive, if one recalls that $\hat{V}(s_t)$ is an estimate of the return the agent expects to receive when starting from state s_t. By the same reasoning, $\hat{V}(s_{t+1})$ is an estimate of the expected return from state s_{t+1}. Thus, there is no need to wait for a sample of the actual return to become available. It is possible to make a useful update immediately, since the observed reward r_{t+1} is available and the discounted return from state s_{t+1} is available in $\hat{V}(s_{t+1})$ (Sutton, 1988).

Sarsa Algorithm

The Sarsa algorithm uses the TD method described above, but instead of learning to estimate the value of states, this algorithm learns to estimate the value of state-action pairs. It follows the idea of GPI described earlier, and is shown in Algorithm 2.1 on the next page. The name "Sarsa" is derived from the fact that the algorithm uses the following signals in every time step:

$$s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}$$

to update the state-value function $\hat{Q}$. The update rule is showed in Algorithm 2.1 on the facing page, line 10. Compare it with (2.6) above.

The agent used for benchmarking in the evaluations was based on the Sarsa algorithm.

2.2 Neurobiology

Functionally, the cells within the brain can be said to be of two fundamentally different types: glial cells and neurons. Glial cells have a strictly supportive functionality, and will not be discussed further. The information processing that occurs in the brain is carried out by the neurons.

Require: $\alpha, \gamma \in [0, 1]$ ▷ Learning (α) and discount (γ) rate parameters
Initialize $\hat{Q}(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize $\hat{Q}(s, a) = 0$ for all $s \in \mathcal{S}^+ - \mathcal{S}, a \in \mathcal{A}$ ▷ $\hat{Q} = 0$ in terminal states
for all episodes **do**
    Observe initial state s_0
    Choose a_0 from s_0 using policy π_0 derived from $\hat{Q}$
    $t \leftarrow 0$
    repeat
        Perform action a_t, receive reward r_{t+1}, observe next state s_{t+1}
        Choose a_{t+1} from s_{t+1} using policy π_t derived from $\hat{Q}$
        $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha(r + \gamma\hat{Q}(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t))$
        $t \leftarrow t + 1$
    until s_t is a terminal state
end for

Algorithm 2.1 Sarsa algorithm. Algorithm adapted from Sutton and Barto (1998, p. 146).

Neurons

Neurons are cells which have become specialized in the generation and transmission of electrical signals. They come in many different shapes and sizes, and yet they all have a similar structure. Morphologically, we can identify four parts: its cell body, dendrites, axon, and its presynaptic terminals. Figure 2.2 on the next page depicts a schematic of a typical neuron in the brain.

The dendrites and the axon are the two types of extensions that emerge from the cell body. The dendrites branch out in a tree-like tendering fashion, while the axon is long, thin, and cylindrical. The initial segment of the axon, called the trigger zone, joins with the cell body. Near its end, the axon branches out into several fine thread-like extensions called axon terminals. The axon terminals end in the presynaptic terminals, marked • in Figure 2.2 on the following page. Presynaptic terminals may come into close proximity with another neuron's dendrites or cell body and form special contact sites known as synapses. These are sites where neural signals can pass from one neuron to another. One such synapse is shown to the top left of Figure 2.2 on the next page.

The output signal of a neuron is known as an action potential (AP), and is generated near the trigger zone. Once generated, it travels along the axon, away from the cell body, and towards the axon terminals. The direction of the AP flow is illustrated in Figure 2.2 on the following page by the dark arrows ➡. AP conduction speed along an axon is quite slow. This is not a problem for short axons, but to enable neural signaling over long distances, some axons are put through a process called myelination (Purves et al., 2004, pp. 63–65). This process causes the axon to become wrapped with a substance called myelin, thereby effectively insulating it. APs are said to be actively propagated along such axons because they are now regenerated, i.e. restored to full strength, at regular intervals at sites known as nodes of Ranvier. AP conduction speed is greatly increased with myelination.

The rate at which a neuron produces APs is loosely referred to as its firing frequency. Following dendritic input, some neurons rarely fire APs while others fire them almost without pausing. It is often useful to classify neurons according to their firing frequency. The current belief is that output from neurons within the cerebral cortex is in fact frequency encoded. In other words, it is the frequency in which APs are fired that is the important information carrier, not the existence or absence of individual APs.

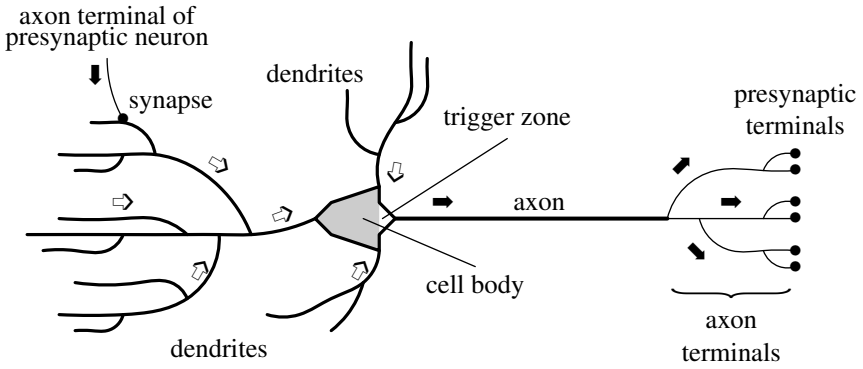


Figure 2.2 Schematic illustration of a typical neuron in the brain. The arrows indicate the flow of neural signals. See the text for more information. Illustration adapted from Gurney (1997, pp. 2, 8) and Haykin (1999, p. 8).

The lower and upper bound of possible frequency values depend on the intrinsic properties of each neuron. See Connors and Gutnick (1990) for a review of different firing patterns.

The arrival of an AP at a presynaptic terminal sets off a chemical sequence of events that result in the production of a neural signal in the dendrites, local to the synapse, of the postsynaptic neuron. However, unlike the AP, this signal is graded, which means that its amplitude can assume a range of values. The so called strength of the synapse determines the amplitude of the postsynaptic neural signal. The change in potential brought about by the arrival of this neural signal is called a postsynaptic potential (PSP). Synapses which transmit neural signals in this manner, i.e. by chemical means, are referred to as chemical synapses.

The signal travels passively, i.e. without regeneration, along the dendrites towards the cell body, and the trigger zone. The direction of this flow is marked with ⇨ in Figure 2.2. Each signal that arrives at the cell body contributes to raising or lowering its potential. In other words, the neural signals are integrated, or "summed together", both spatially and temporally, in the cell body. If the potential is raised above the neuron's so called threshold potential, the neuron will generate an AP.

Learning and Long-Term Memory

This section reviews plausible neural substrates for learning and long-term memory.

Current opinion is that learning is, at least in part, accomplished by means of synaptic plasticity. Synaptic plasticity refers to chemical synapses' ability to alter their strength. Recall that the strength of a synapse determines the amplitude of the PSPs it produces. Purves et al. (2004) point out that it is likely that all chemical synapses are capable of plastic change. Learning may also be accomplished by means of the formation of new synaptic connections, but it is mainly attributed to the alteration of synaptic strength.

The general agreement is that long-term memory depends, at least in part, on long-term changes in the efficacy among relevant synaptic connections (Purves et al., 2004). Biological support for this has been detected in the phenomenon known as long-term potentiation (LTP), first reported by Bliss and Lomo (1973). The effect of LTP is to alter the strength of individual synapses for an extended period of time. The interested

reader is referred to Bliss and Collingridge (1993); Goosens and Maren (2002) for more information about LTP.

As LTP occurs at individual synapses, the information needed to determine the change in synaptic strength must be locally available at each synapse. Information about the presynaptic AP is clearly available, since it induces a PSP in the post-synaptic neuron. Also, evidence is mounting to support the idea that APs are actively back-propagated into the dendritic tree (Magee and Johnston, 1997; Markram et al., 1997; Stuart and Sakmann, 1994). This is in contradiction with the traditional view that dendrites are electrically passive structures whose only purpose is to allow neural signals to travel toward the cell body. It implies that information about the AP of the post-synaptic neuron could also be available at the synapse. Furthermore, it has been shown that this back-propagating AP is of importance in the induction of LTP (Magee and Johnston, 1997). To describe the biology behind this is beyond the scope of this thesis. See Paulsen and Sejnowski (2000), and Linden (1999) for more information about the role of back-propagating APs in the induction of LTP.

Reinforcement Learning in the Brain

This section will discuss aspects of neurobiological theory and research that relate to the field of RL.

There is support for the idea that the brain maintains an internal representation of which state is currently occupied in the environment. In rodents it has been shown that there are neurons in the hippocampus, a structure deep within the brain that is vital for memory and emotional behavior (Purves et al., 2004), which fire APs only when the animal occupies certain spatial locations. For this reason, these neurons are called place cells (Purves et al., 2004, p. 584). The restricted portion of the environment a place cell responds to is called its place field (Foster et al., 2000, p. 1).

The normalization model proposed in Carandini et al. (1997) provides clues as to how the brain might be able to compute a probability mass function, and consequently a policy. The idea is that cells may be contained in a so called normalization pools. The effect is that neurons in such a pool will inhibit each other, thereby undergoing "normalization". Their model included a parameter called the gain, which could be used to control the rate of exploration versus exploitation. Another article of interest was written by Uscher et al. (1999). It too presented a model where neurons inhibited each other. Their model had a mechanism for controlling the rate of exploration versus exploitation as well. The model was able to explain their experimental data.

O'Doherty et al. (2003) reported of a convincing experiment which suggests that TD learning is expressed in the human brain. Specifically, they found that the output of a TD learning algorithm was able to account for responses given by their human test-subjects. Schultz et al. (1997) presented a review of experimental evidence which supports this idea. They concluded that learning may indeed be driven by changes in expectations about future rewards and punishments. They found that the TD algorithm was well suited for understanding the experimental data. The same ideas were also expressed by Montague et al. (1996).

2.3 Artificial Neural Networks

There is no universally accepted definition of what an artificial neural network (ANN) is. In this thesis we adopt the following definition by Gurney (1997, p. 1):

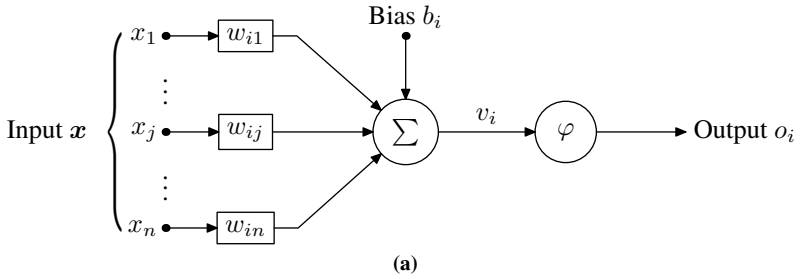


Figure 2.3 Artificial neuron i. Compare with Figure 2.2 on page 10.

> "A neural network is an interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on the animal neuron. The processing ability of the network is stored in the inter-unit connection strengths, or weights, obtained by a process of adaptation to, or learning from, a set of training patterns."

Note that it is common to drop the word artificial, if it is understood that the neural network is artificial. These "artificial neurons" will henceforth be referred to simply as units. Typically, such a unit is modeled as in Figure 2.3. A unit is able to receive a set of signals. The particular unit i shown in Figure 2.3 receives signals $\boldsymbol{x} = (x_1, \ldots, x_n)$ from n so called input nodes, depicted as $\bullet$. Signals from input nodes will be referred to as input signals. The only purpose of the input nodes is to provide input signals to one or more units.

The input/output behavior of a unit, such as i, is as follows: Each incoming signal to the unit is multiplied with its corresponding weight. These weights are shown in boxes in Figure 2.3. For example, input signal x_j is multiplied with weight w_{ij}. Note the order of the subscripts of weight w_{ij}. The convention in this thesis is to write the index of the target unit i first, then the index of the source node/unit j. Typically one allows the weights to assume any real value. The weighted input signals, for example $w_{ij}x_j$, are then summed together to form the so called induced local field $v_i = b_i + \sum_{j=1}^{n} w_{ij}x_j$. Also included is a bias term b_i. It is common practice to treat the bias b_i as just another weight $w_{i0} = b_i$ which just happens to be connected to an input signal x_0 that is always one: $x_0 = 1$. This lets us rewrite the equation for the induced local field v_i as

$$v_i = \sum_{j=0}^{n} w_{ij}x_j \tag{2.7}$$

Next, the induced local field signal v_i is passed to the transfer function φ. There is a wide variety of transfer functions commonly used, and one must choose an appropriate one on a task-per-task basis. It may, for example, be entirely linear, in which case the output o_i from unit i is $o_i = \varphi(v_i) = v_i$. If we wish to model the all-or-none property of the AP we could instead choose a threshold function. Such a threshold function could for example output 1 if $v_i > \theta$, corresponding to the firing of an AP, and output 0 otherwise. In this example, θ corresponds to the threshold potential of the animal neuron modeled. Recall that it is believed that the output from a neuron in the cerebral cortex is frequency encoded, with an upper and a lower bound on the frequency. This could be modeled by having φ map the induced local field v_i into the continuous interval $[0, 1]$. This interval would then represent the whole range of AP

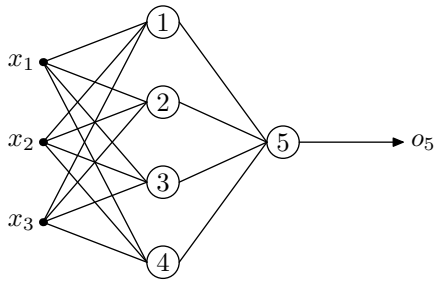


Figure 2.4 Example of a two-layered, feedforward, fully connected ANN.

firing frequencies, from the lowest (0) to the highest (1), the neuron could potentially produce.

According to the definition, an ANN is an interconnected assembly of units like the one modeled in Figure 2.3 on the preceding page. Units can be connected in infinitely many different ways, but some network structures have been found to be more useful than others. An example of how the units could be connected is shown in Figure 2.4. This is an example of a two-layered, feed-forward, fully connected ANN. The units are arranged in two layers. Units one through four reside in layer one, and unit five alone occupies the second layer. The last layer is often referred to as the output layer. The layers in between the input nodes and the output layer are sometimes called hidden layers. Signals propagate forward, from left to right without any feedback loops. It is furthermore classified as a fully connected ANN, because every unit is connected to every other unit/node in the preceding layer. Each connection, both node-to-unit and unit-to-unit, is modulated by a weight. When depicting the whole network structure, as in Figure 2.4, these weights are usually not explicitly illustrated.

The input/output behavior of the network as a whole is computed on a layer-per-layer basis. That is, first the output of layer one is computed using the input signals from the input nodes. The output signals from the first layer then serves as input to the second layer, and so on until the output signals from the output layer has been computed. These output signals constitute the output from the network.

According to the definition, the ANN obtains its processing ability by adjusting its weights. This is often an iterative process, during which the weights of the network are updated according to the chosen learning rule. A popular choice for training multilayer[2] feed-forward ANNs, such as the one illustrated in Figure 2.4, is the error back-propagation algorithm, or some variation of it. The idea is to first present the network with a pattern of input signals $\boldsymbol{x}$ at the input nodes. Next, these signals are propagated through the network, layer by layer, until they reach the output layer. The output from the units in the output layer is compared with the desired output, and an error signal is computed. The desired output for each pattern of input signals is known beforehand, as it is a supervised learning algorithm. The computed error is then propagated backwards through the network, and the weights are adjusted in such a way that the error will be smaller the next time the same pattern of input signals is presented to the network. The details of the algorithm are described in Haykin (1999, pp. 156–175) and Gurney (1997, pp. 65–69).

[2] A multilayer ANN has at least two layers of units.

Chapter 3

Biologically Plausible Models of Learning

This chapter presents a review of a selection of what was considered to be biologically plausible models of learning. Recall that a model was defined to be biologically plausible if it could be implemented with an ANN that used a biologically plausible weight update rule. What was considered to be a biologically plausible weight update rule is discussed in Section 4.2 on page 22.

Some of the original notation has been changed to better comply with the notation used in this thesis.

3.1 S-model A_{R-P} Weight Update Rule

The S-model A_{R-P} weight update rule[1] was first introduced by Barto and Jordan (1987). They used it in a supervised learning task to update the weights and biases of the units in the hidden layer in a two-layered feed-forward ANN.

Using the notation introduced with Figure 2.3 on page 12, the input/output behavior of the units in the hidden layer were

$$o_i = \varphi(v_i) = \begin{cases} 1 & \text{with probability } f(v_i) \\ 0 & \text{with probability } 1 - f(v_i), \end{cases} \tag{3.1}$$

where $f(v_i) = 1/(1 + e^{-v_i})$. The function f can be recognized as the logistic sigmoid function, whose range is the continuous interval $[0, 1]$.

As it was a supervised learning task, the desired output from the ANN was known. The weights to the units in the output layer were updated according to the error back-propagation algorithm (see Section 2.3 on page 13). However, the units in the hidden layer were updated differently. Based on the desired output and the actual output, a single numerical measure of success $r \in [0, 1]$ was computed. Their weights and biases were then updated according to

$$\Delta w_{ij} = \rho \left[r(o_i - f(v_i)) + \lambda(1 - r)(1 - o_i - f(v_i)) \right] x_j, \tag{3.2}$$

[1] The S-model A_{R-P} rule is an extension of the A_{R-P} algorithm introduced by Barto and Anandan (1985).

where $\rho > 0$ and $\lambda \in [0, 1]$ are constants. The numerical measure of success r was required to reside in the interval $[0, 1]$, where 0 and 1 meant complete failure and complete success respectively. In other words, if $r = 1$, the output from the ANN matched the desired output perfectly.

The weight update rule in (3.2) can be understood by considering the extreme cases of $r = 1$ and $r = 0$. In the case of $r = 1$, i.e. we have a complete success, the right-hand side reduces to $\rho\,[o_i - f(v_i)]\,x_j$. This would modify the weight w_{ij} so that whatever $o_i \in \{0, 1\}$ the unit selected, the likelihood of it selecting the same o_i again would increase. Conversely, in the case of complete failure we have $r = 0$, and the weight w_{ij} would instead be adjusted so that this likelihood was decreased. No weight change at all took place if $x_j = 0$, since the weight then did not contribute to the choice of o_i.

The authors stated that ρ controlled the magnitude of the weight change, and λ: "determines the degree of asymmetry in the magnitude of the weight change".

Bartlett and Baxter (1999) used a similar update rule for which a great deal of theory has been worked out. By setting $\lambda = 0$ in (3.2), it corresponds to a specific instance of their update rule: the one obtained by setting their quantity $\beta = 0$. For a biologically relevant application of the S-model A_{R-P} rule, see Mazzoni et al. (1991).

3.2 Dual Projection BCPNN RL System

BCPNN RL systems (Johansson and Lansner, 2002b; Johansson et al., 2003) are systems based on the Bayesian Confidence Propagating Neural Network (BCPNN), and are designed to solve RL problems. Previous work have studied BCPNNs in the context of classification (Holst, 1997; Holst and Lansner, 1993), classical conditioning (Johansson and Lansner, 2002a), and as a model for memory (Johansson et al., 2001; Sandberg et al., 1999).

Johansson and Lansner (2002b) described three BCPNN RL systems of varying complexities and capabilities. All were based on the concept of populations and projections. A population can be thought of as a set of units. All three systems utilized two populations of units. The first population, the state population, represented the states. It contained one unit for every non-terminal state in the environment. These units will be referred to as state units. The other population, the action population, represented the actions available to the agent. The authors assumed that the same set of actions were available to the agent in all non-terminal states. Therefore, this population contained as many units as there were possible actions. Units in the action population will be referred to as action units. A projection was defined as: "the computation needed to derive the weights and biases of the connections between two populations". One projection corresponded to a full set of connections between the units in the two populations. In other words, there was one connection from every state unit to every action unit, for each of the two projections.

The system of interest for this project was their dual projection BCPNN RL system. This system had features added to be able to handle negative rewards. It also possessed the ability to relearn, i.e. it was able to adapt to a changing environment. This was made possible by the use of two projections: one "positive" $(+)$ and one "negative" $(-)$. The positive projection enhanced correlations between the units in the two populations, while the negative projection inhibited them.

Some notation needs to be introduced. The output from unit j in the state population was denoted x_j. The weight from state unit j to action unit i was denoted $w_{ij}^{\pm}$,

for the two $(\pm)$ projections respectively. The bias value of action unit i was denoted $\beta_i^{\pm}$. Every action unit had a support value h_i, which represented the strength of the network's "belief" in that unit. There was also one support value for each action unit for both projections, denoted $h_i^{\pm}$. The number of state and action units are denoted n and m respectively.

The input/output behavior of the system as a whole was as follows. First the unit representing the current state in the state population was activated. Secondly, the support values $h_i^{\pm}$ for both projections, for every action unit i, were set to zero: $h_i^{\pm} = 0$. These projection support values were then updated by iterating

$$\Delta h_i^{\pm} \leftarrow \left(\log(\beta_i^{\pm}) + \log\left(\sum_{j=1}^{n} w_{ij}^{\pm} x_j \right) - h_i^{\pm} \right) / \tau_C \tag{3.3}$$

until stability, where $\tau_C \in [1, \infty[$ was a constant. The actual support values for the action units were then simply

$$h_i = h_i^{+} - h_i^{-}. \tag{3.4}$$

These action support values were then normalized by

$$p_i = \frac{e^{Gh_i}}{\sum_{k=1}^{m} e^{Gh_k}} \tag{3.5}$$

to obtain the (partial) policy used for action selection, where $G \in [0, \infty[$ was a constant. Thus, p_i was the probability of choosing action i.

After having chosen an action the system received a reward r; all according to the agent-environment interaction described in Section 2.1 on page 5. The activities, S_i and S_j, of the action and state units were clamped to represent the action, and the state in which the action was chosen. The activities were set to one for the state and action unit in question. The other units' activities were set to zero. The activities were used to update the so called trace (E) variables according to

$$\begin{aligned} \Delta E_i^{\pm} &= (S_i - E_i^{\pm})/\tau_E \\ \Delta E_j^{\pm} &= (S_j - E_j^{\pm})/\tau_E \\ \Delta E_{ij}^{\pm} &= (S_i S_j - E_{ij}^{\pm})/\tau_E, \end{aligned} \tag{3.6}$$

where $\tau_E \in [1, \infty[$ was a constant. The trace variables were updated regardless of the value of the reward r. The next step, however, only took place when the reward assumed a non-zero value: $r \neq 0$. It involved updating the so called memory (P) variables according to

$$\begin{aligned} \Delta P_i^{\pm} &= \kappa(E_i^{\pm} - P_i^{\pm})/\tau_P \\ \Delta P_j^{\pm} &= \kappa(E_j^{\pm} - P_j^{\pm})/\tau_P \\ \Delta P_{ij}^{\pm} &= \kappa(E_{ij}^{\pm} - P_{ij}^{\pm})/\tau_P, \end{aligned} \tag{3.7}$$

where $\tau_P \in [1, \infty[$ was a constant, and κ was the so called print-now signal. If $r > 0$, then the positive $(+)$ projection was updated with print-now signal $\kappa = r$ and the negative projection $(-)$ was decayed, and vice versa with $\kappa = -r$ if $r < 0$. That way, whichever projection was updated, $\kappa > 0$. Decaying a projection meant updating the memory variables according to

$$\begin{aligned} \Delta P_i^{\pm} &= (1/m - P_i^{\pm})/\tau_P \\ \Delta P_j^{\pm} &= (1/n - P_j^{\pm})/\tau_P \\ \Delta P_{ij}^{\pm} &= (1/mn - P_{ij}^{\pm})/\tau_P, \end{aligned} \tag{3.8}$$

instead of using (3.7). Finally the weights and biases were computed as

$$\beta_i^{\pm} = P_i^{\pm} + \lambda_0 \quad \text{and} \quad w_{ij}^{\pm} = \frac{P_{ij}^{\pm} + \lambda_0^2}{(P_i^{\pm} + \lambda_0)(P_j^{\pm} + \lambda_0)}, \tag{3.9}$$

where $\lambda_0 > 0$ was constant whose purpose was to prevent the possibility of taking the logarithm of zero in (3.3). The authors chose $\lambda_0 = 10^{-4}$ in their experiments.

Initially, before learning began, the trace and memory variables were set according to

$$\begin{aligned} E_i &= P_i = 1/m \\ E_j &= P_j = 1/n \\ E_{ij} &= P_{ij} = 1/mn. \end{aligned} \tag{3.10}$$

On a biological note, they stated that the trace (E) variables were thought to correspond to the influx of calcium in a synapse. Also, the memory (P) variables were intended to correspond to LTP in a synaptic coupling. The print-now signal κ was thought to correspond to release of inter-cellular neuromodulator substances, such as dopamine. See also Wahlgren and Lansner (2001).

3.3 Doya's Basal Ganglia Loop Hypothesis

The basal ganglia loop[2] is believed to play a major role in RL (Doya, 1999). This section presents a set of hypothesizes about possible roles of its components with respect to RL. The hypothesis of most interest to this project was presented by Doya (2002). It assumed the existence of two sets of basis functions:

$$b_j(s) \quad \text{and} \quad c_k(s, a), \tag{3.11}$$

where s and a denoted state and action respectively. The basis functions provided the internal representation of states and actions, and were assumed to reside within the cerebral cortex.

Here is a description of how actions were assumed to be selected, according to the hypothesis. The basis functions $c_k(s, a)$ projected, modulated by weights w_k, into the so called matrix compartment of the basal ganglia's input zone, where state-action values $Q(s, a)$ were formed:

$$Q(s, a) = \sum_k w_k c_k(s, a). \tag{3.12}$$

These state-action values were projected into the substantia nigra pars compacta and the globus pallidus. There they were subjected to competitive dynamics to realize a probability distribution over the actions, and subsequently an action selection. It was suggested that the probability distribution could be computed with

$$p_i = \frac{e^{\beta Q(s,a_i)}}{\sum_{j=1}^{m} e^{\beta Q(s,a_j)}}, \tag{3.13}$$

[2] It is beyond the scope of this thesis to describe the basal ganglia loop. For simplicity, one can think of it as a set of projections that lead from the cerebral cortex to the basal ganglia, to the thalamus, and back again. See Purves et al. (2004, pp. 417–434) for more information.

where β was a constant parameter, and m was the number of candidate actions. In words, p_i denoted the probability of choosing action i. Using the distribution in (3.13), an action was selected, projected to the thalamus, and then projected back to the cerebral cortex to close the loop.

The other set of basis functions $b_j(s)$ projected into the so called patch compartment of the basal ganglia's input zone. This projection was modulated by weights v_j, and formed a state value function

$$V(s) = \sum_j v_j b_j(s). \tag{3.14}$$

Using that value function, a TD error signal $\delta(t)$ could be computed in the the substantia nigra dopaminergic neurons (Doya, 2000a) according to

$$\delta_t = r_t + \gamma V(s_t) - V(s_{t-1}), \tag{3.15}$$

where γ was a constant parameter. It may look awkward, but (3.15) can be rewritten in way which could potentially be implemented within the basal ganglia loop (Doya, 2000c). The TD error signal δ_t was then used to update the weights v_j and w_k according to

$$\begin{aligned} \Delta v_j &= \alpha \delta_t b_j(s_{t-1}) \\ \Delta w_k &= \alpha \delta_t c_k(s_{t-1}, a_{t-1}), \end{aligned} \tag{3.16}$$

where α was a constant parameter.

Another hypothesis by Doya (2000b) attempted to provide a biological relevance to the above mentioned parameters/signals: δ, γ, β, and α. Specifically, it stated that: (i) dopamine signals the TD error δ (which we have already mentioned), (ii) serotonin controls the discount factor γ, (iii) noradrenaline controls the inverse temperature β, and (iv) acetylcholine controls the learning rate α.

3.4 Water-Maze Navigation Model

This model was presented by Foster et al. (2000). It was intended to show: "how hippocampal place cells might be used for spatial navigation". Specifically, it was intended to model the behavior of rats in water-maze tasks.

The place cells were modeled with Gaussian functions. Assuming the rat was at position p, the output from place cell j was computed as

$$f_j(p) = e^{-\|p - s_j\|^2 / 2\sigma^2}, \tag{3.17}$$

where s_j was the center location of the cell's place field, and σ was the breadth of the field. It can be seen that the place cell will respond the strongest when $p = s_j$, i.e. when the rat was in the center of the cell's place field.

The output from the place cells projected to two computational sub-structures: the actor and the critic. The purpose of the actor was to produce actions, while the critic's job was to criticize the actions selected by the actor. At each time step, the actor had to choose one of the eight actions shown in Figure 3.1 on the next page: north (N), north-east (NE), east (E), and so on. Each action was represented with a corresponding unit i inside the actor. All place cells projected to these so called action units, modulated by weights. The weight between place cell j and action unit i was denoted z_{ij}. The

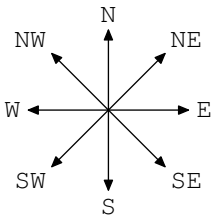


Figure 3.1 Actions available in the water-maze tasks. Each action was represented with a so called action unit i.

critic contained a single unit C. All place cells projected to that unit, also modulated by weights. The weight to unit C from place cell j was denoted w_j.

The actor selected an action in the following way. First, the support value h_i of each individual action cell i was computed by

$$h_i(p) = \sum_{j=1}^{n} z_{ij} f_j(p), \tag{3.18}$$

where n denoted the number place cells. The authors interpreted these support values h_i as the relative preference for swimming in the ith direction at location p. The actual direction chosen was selected according to the probability distribution

$$p_i = \frac{e^{2h_i}}{\sum_k e^{2h_k}}, \tag{3.19}$$

in other words p_i was the probability of swimming in direction i.

Learning was based on the output from the critic's unit C, whose output was computed simply as

$$C(p) = \sum_{j=1}^{n} w_j f_j(p). \tag{3.20}$$

The next step was to compute the so called prediction error δ_t

$$\delta_t = r_t + \gamma C(p_{t+1}) - C(p_t), \tag{3.21}$$

where r_t was the reward given to the RL agent at time step t, and γ was a constant parameter. Positions p_{t+1} and p_t were the positions occupied by the agent at time steps $t+1$ and t respectively. Finally, the weights of the model were updated according to

$$\begin{aligned} \Delta w_j &\propto \delta_t f_j(p_t) \\ \Delta z_{ij} &\propto \delta_t f_j(p_t) g_i(t), \end{aligned} \tag{3.22}$$

where $g_i(t)$ was one if action i was chosen at time step t, and zero otherwise.

Chapter 4

Method of Evaluation

This chapter describes how the goal of this project was approached. The evaluation was carried out by having the agents solve a set of challenging tasks. Recall that a task is a complete specification of a RL environment, as was explained in Section 2.1 on page 6. The chapter begins by listing some features all tasks had in common. Next comes a discussion of what was considered reasonable to require of a biologically plausible weight update rule. This motivates the selection of biologically plausible models of learning presented in Chapter 3. The models were turned into RL agents by fitting them into a framework, which is described in Section 4.3 on the next page. The evaluation procedure is described in Section 4.4 on page 25, and the tasks themselves are described in the last section.

As was mentioned in Chapter 1, due to time restrictions only two models were evaluated: the S-model A_{R-P} weight update rule (see Section 3.1 on page 15), and the dual projection BCPNN RL system (see Section 3.2 on page 16). They were quite different models of learning, which made it interesting to compare them. The S-model A_{R-P} weight update rule only specified the input/output behavior of the units themselves, and how to update weights and biases. In contrast, the dual projection BCPNN RL system was pretty much a complete RL agent in itself.

4.1 Task Features

The tasks all shared some important features which will be reviewed in this section.

First of all, all actions were available to the agent in every non-terminal state $s \in \mathcal{S}$:

$$\mathcal{A} = \mathcal{A}(s). \tag{4.1}$$

Intuitively, this was not an unreasonable assumption. After all, in real life one is free to choose any action in any situation. Admittedly, some choices are better than others.

Secondly, all tasks were restricted to having a finite set of actions and states, with only one terminal state. The number of non-terminal states and actions were denoted n and m respectively:

$$|\mathcal{S}| = n < \infty \quad \text{and} \quad |\mathcal{A}| = m < \infty. \tag{4.2}$$

Since there was only one terminal state, the total number of states was: $|\mathcal{S}^+| = n + 1$.

Thirdly, the reward signal r_t was restricted to the continuous interval $[-1, +1]$ for all time steps t:

$$r_t \in [-1, +1]. \tag{4.3}$$

This ought not to have been a serious limitation. Any task can be adjusted to hand out rewards in this interval by dividing all positive rewards with the maximum positive reward, and vice versa for the negative rewards. Biologically, a reward of $+1$ could correspond to a sense of complete bliss, while a reward of -1 could correspond to the worst pain imaginable. In between those two extremes was, of course, the indifferent reward ± 0.

4.2 Biologically Plausible Weight Update Rule

A biologically plausible weight update rule should only use information expected to be present at a biological synapse. The output from the presynaptic neuron is certainly available, as it arrives in the presynaptic terminal in the form of an AP. There is strong evidence supporting the idea of back-propagating APs, see Section 2.2 on page 10, which would mean that the output from the postsynaptic neuron is also available at the synapse. Also, we would expect such an update rule to be relatively "simple". This, admittedly vague statement, is meant to reflect the fact that a biological synapse cannot be expected to perform complicated arithmetic. At least, this author knows of no biological data supporting this idea.

The error back-propagation algorithm (see Section 2.3 on page 13), or some variation of it, is perhaps the most commonly used method of training ANNs today. There are many reasons this is not considered to be biologically plausible. For one, it is a supervised learning method and thus requires the desired output to be known. Secondly, it requires the error to be propagated backwards through the network. Biologically, this would for example mean that a signal (the error) has to travel from the presynaptic terminals, and back up the axon. There is no biological support for this. Thirdly, the error back-propagation algorithm requires the computation of a staggering number of partial derivatives; namely one for every weight in the network. This contradicts the assumption that the weight update rule should be relatively simple.

4.3 RL Agent Framework

The models of learning were fitted to the same RL agent framework, see Figure 4.1 on the next page, to allow for a more intuitive evaluation and comparison. This section will describe that framework, and how each model was fitted into it.

The framework consisted of three stages: the ANN stage, the normalization stage, and the selection stage. Signals propagated from left to right, starting with the input nodes.

The input nodes, each depicted with a • in Figure 4.1 on the facing page, provided unary input $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ to the ANN stage. The restriction to use unary input was put in place in order to simplify the evaluations. The framework itself allows for arbitrary input. The vector $\boldsymbol{x}$ was called the state vector, because it represented the state of the environment. Each of its elements represented one of the $n < \infty$ possible states. Assuming the environment was in state k, then $x_k = 1$ and all other elements in the vector were zero.

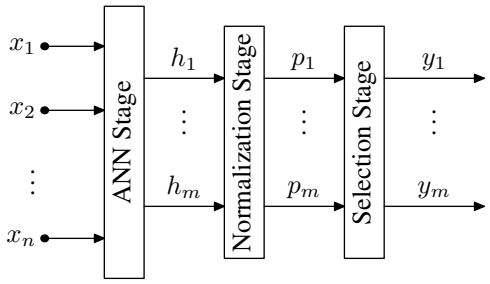


Figure 4.1 The three stages of the RL agent framework.

The ANN stage contained the model specific algorithms. Its output was called the support vector $\boldsymbol{h} = (h_1, \ldots, h_m)$, since it contained support values for each of the $m < \infty$ actions. Each support value $h_i \in \mathbb{R}$ represented the model's support for that particular action.

The support vector $\boldsymbol{h}$ was fed into the normalization stage, which computed a probability distribution $\boldsymbol{p} = (p_1, \ldots, p_m)$ according to

$$p_i = \frac{e^{Gh_i}}{\sum_{k=1}^{m} e^{Gh_k}}, \tag{4.4}$$

where $G \in [0, \infty[$ was the gain parameter; so called because it would favor the actions with higher support values as $G \to \infty$. Each element p_i in $\boldsymbol{p}$ thus gave the probability of selecting action i. The probability distribution $\boldsymbol{p}$ was not a complete policy π, but at least it allowed the agent to choose its next action. All agents had to be able to do this, so it was quite natural to include it in the framework. The normalizing function used in (4.4) is called the Gibbs (or Boltzmann) distribution. It was a natural choice, since it was common to many of the models.

The effects of the gain parameter can clearly be seen by considering the special case $m = 2$. In this case, (4.4) for $i = 1$ becomes $p_1 = e^{Gh_1}/(e^{Gh_1} + e^{Gh_1})$, which can be simplified to $p_1 = 1/(1 + e^{-G(h_1 - h_2)})$. This can be recognized as the logistic function with slope parameter G. Figure 4.2 shows the effect of different values of G

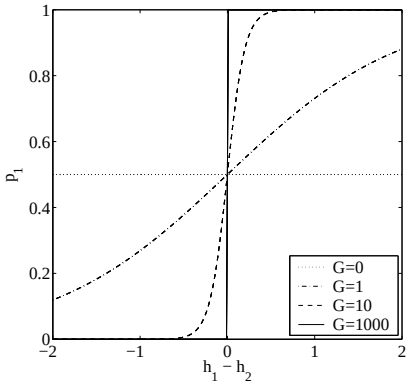


Figure 4.2 Effects of different gain parameter G values on the Gibbs distribution when selecting between two actions. Inspired by Doya (2002, p. 502).

for the case $m = 2$. When G becomes very large, even a small difference in support values ($h_1 - h_2$) will have a big impact on the policy. Conversely, when $G = 0$, the policy is completely stochastic, regardless of the support values. Presumably, G can be used to control the trade-off between exploration ($G = 0$) and exploitation ($G = \infty$), see Section 2.1 on page 7. For more information on the Gibbs distribution, see Sutton and Barto (1998, p. 30–31).

The probability distribution $\boldsymbol{p}$ was then fed into the last stage, the selection stage, where an action was chosen. The chosen action was represented by the unary vector $\boldsymbol{y} = (y_1, \ldots, y_m)$, where each element represented an action. Thus, if action i was chosen, then $y_i = 1$ and all other elements were zero.

Sarsa

This agent was based on the Sarsa algorithm listed in Algorithm 2.1 on page 9. In order to be of maximum use for benchmarking, it was implemented similarly to the biologically plausible agents.

The state-action values $\hat{Q}$ were initialized according to $\hat{Q}(s, a) = 1/mn$ for all non-terminal states $s \in \mathcal{S}$ and all actions $a \in \mathcal{A}$. This resembles the initialization of the dual projection BCPNN RL System.

In every time step t, the policy π_t was obtained by normalizing all m state-action values where the state was s_t, namely: $\{\hat{Q}(s_t, a)|a \in \mathcal{A}\}$. The state-action values were normalized by using the Gibbs distribution in (4.4), similarly to the other agents.

A_{R-P} Agent

This agent was based on the S-model A_{R-P} weight update rule in Section 3.1 on page 15. For simplicity, the agent was named the A_{R-P} agent.

The ANN stage of this agent was a one-layer, feedforward, fully connected ANN. Its input layer consisted of input nodes which simply relayed the state vector $\boldsymbol{x}$. Its output layer contained units with the input/output behavior specified in (3.1). There were m units in this layer, one for every action. The output from the units in this layer was the support vector $\boldsymbol{h}$ in Figure 4.1 on the previous page.

The S-model A_{R-P} weight update rule was designed to work with a numerical measure of success $r \in [0, 1]$, but these evaluations nevertheless allowed the reward to range between -1 and $+1$, see (4.3). The reward was not rescaled into the interval $[0, 1]$ to accommodate the A_{R-P} agent. Because of this, the range of the output from the ANN stage in the framework, i.e. the support vector $\boldsymbol{h}$, was unknown. The normalization stage ensured that a valid action selection still took place. The A_{R-P} agent was expected to display poor performance on tasks with negative rewards, as it lacked the machinery to cope with it.

Initially, the weights were set according to $w_{ij} = 1/mn$ and the biases according to $b_i = w_i 0 = 1/m$. This initialization was chosen because it resembles the way the Sarsa and the BCPNN agents are initialized.

BCPNN Agent

This agent was based on the dual projection BCPNN RL system presented in Section 3.2 on page 16. That system was pretty much a complete RL agent in itself. As it was fitted into the framework it was only slightly simplified.

The state population simply contained input nodes which relayed the state vector $\boldsymbol{x}$, same as the A_{R-P} agent.

The activities S_i and S_j used in equation (3.6) were set to $S_i(t) = y_i(t-1)$ and $S_j(t) = x_j(t-1)$, where explicit use of time t was made in order to be able to specify that the activities should represent the last action taken and in what state it was taken, respectively. This meant the action taken, and the state occupied, at time step $t-1$.

For simplicity it was decided to lock the two parameters τ_C and λ_0 to: $\tau_C = 1$ and $\lambda_0 = 10^{-4}$. Having $\tau_C = 1$ meant there was no need to iterate (3.3). The projection support values could now be computed in one go. This left two parameters to investigate in the evaluations: τ_E and τ_P.

The normalization in (3.5) was taken care of by the normalization stage in Figure 4.1 on page 23.

Finally, note that (3.3) now could be written

$$\Delta h_i^{\pm} = \log(\beta_i^{\pm}) + \log(w_{ik}^{\pm}) = \log(\beta_i^{\pm} w_{ik}^{\pm}), \tag{4.5}$$

where k corresponded to the input node of the currently visited state. This simplification was possible because $\tau_C = 1$, as was explained above, but also because $\boldsymbol{x}$ was a unary vector, which meant that the sum $\sum_j w_{ij}^{\pm} x_j$ was reduced to $w_{ik}^{\pm}$. Recall that, since $\boldsymbol{x}$ was a unary vector, we had $x_j = 0$ for all $j \neq k$.

4.4 Evaluation and Performance Measure

Agents were evaluated according to Algorithm 4.1 on the following page. It shows how a performance measure called the episode value $u_{ij} \in [0, 1]$ was collected after the completion of every episode in every run. The episode value served as a measurement of how well the agent performed in that episode. An episode value of one indicated that the agent found the optimal solution. Conversely, $u_{ij} = 0$ meant that the agent failed completely. See the description of each task in Section 4.5 on the next page for information on how the episode values were computed for each task.

The episode values u_{ij} were gathered in a $R \times E$ matrix $\boldsymbol{U}$:

$$\boldsymbol{U} = \begin{bmatrix} u_{11} & \dots & u_{1j} & \dots & u_{1E} \\ \vdots & & \vdots & & \vdots \\ u_{i1} & \dots & u_{ij} & \dots & u_{iE} \\ \vdots & & \vdots & & \vdots \\ u_{R1} & \dots & u_{Rj} & \dots & u_{RE} \end{bmatrix} \tag{4.6}$$

Note that a row i in matrix $\boldsymbol{U}$ contained the episode values collected in run i. The following approach was used to summarize this data. A single numeric measure u_i of the performance of the agent in run i was obtained by computing the mean, taken over all episodes in that run:

$$u_i = \frac{1}{E} \sum_{j=1}^{E} u_{ij} \tag{4.7}$$

This measure u_i will be referred to as the run value of run i. These run values were collected in a vector $\boldsymbol{u} = (u_1, \dots, u_i, \dots, u_R)$. The data in $\boldsymbol{u}$ was then in turn summarized by computing the well-known five number summary: minimum value $u_{\min}$, lower quartile Q_1, median Q_2, upper quartile Q_3, and maximum value $u_{\max}$. The

Require: $\infty > E, R \in \mathbb{N}^+$ ▷ Number of episodes (E) and runs (R)
for $i \leftarrow 1, 2 \ldots, R$ **do**
  Agent is initialized
  for $j \leftarrow 1, 2, \ldots, E$ **do**
    Agent observes initial state s_0
    $t \leftarrow 0$
    repeat
      Agent performs action a_t
      Agent receives reward r_{t+1}
      Agent observes next state s_{t+1}
      $t \leftarrow t + 1$
    until s_t is a terminal state
    $u_{ij} \leftarrow$ value of episode
  end for
end for

Algorithm 4.1 Evaluation algorithm.

minimum value $u_{\min}$ was the worst-case performance of the agent. That value was frequently used in the evaluations, since it provided a lower bound on the agent's performance. Conversely, the maximum value $u_{\max}$ was the best-case performance of the agent. The difference $u_{\max} - u_{\min}$ was sometimes used to estimate the stability of the agent's performance. Box and whisker plots were generated for especially interesting results, where outliers were drawn with asterisks $(*)$. See Tamhane and Dunlop (2000, pp. 114, 121–123) for more information about the statistical concepts mentioned here.

In order to ensure a reliable result, agents were evaluated over 2000 runs. Thus, $R = 2000$ in all evaluations. The number of episodes ranged from 10 up to 2050, depending on the type of evaluation performed.

Using the Gibbs distribution in the normalization stage was a natural choice, but it introduced an extra parameter: the gain G. Before the agents could be properly evaluated, a suitable value of G had to be found. A suitable value would, ideally, not favor either agent.

Empirical Parameter Search

The performance of each agent could be adjusted by changing its parameters. This introduced a difficulty, since it was not obvious how to set the parameters in order to ensure a fair comparison. Too keep it reasonably fair, an empirical search for the optimal parameter setting was conducted for all agents on all tasks. The procedure outlined above was used to obtain the $u_{\min}$ values for a particular parameter setting. This was repeatedly done for a selection of parameter settings. The results were recorded into a table, from where the best $u_{\min}$ value could be read off. The parameters used to obtain it was considered to be optimal. The optimal parameter setting of each agent, on that task, was then used to create box and whisker plots for use in the comparisons.

4.5 Tasks and Challenges

Here follows descriptions of the tasks used in the evaluations. The descriptions include a motivation as to why the task was included. All tasks except one, the 2-armed bandit

which served as a reference task, presented the agents with specific challenges. These challenges were designed to show the existence, or absence, of specific features in the agents, like the ability to handle delayed reward for instance. The tasks were otherwise quite simple in design, in order to allow for an intuitive understanding of the results. All tasks were episodic, and evaluations took place in discrete time with discrete time steps, see Section 2.1 on page 5.

2-Armed Bandit

The 2-armed bandit (2-AB) task was similar to a Las Vegas slot machine: the so called one-armed bandit. However, the bandit used in these evaluations had not one lever, but two. The agent was to select one of the arms, pull it, and receive the pay-off, or reward, from the bandit. The agent was then free to select again. The task had two states:

- state 1 which represented the state of being able to choose an arm, and
- state 2 which represented the state of having pulled an arm.

State 2 was thus a terminal state, and marked the end of the episode. The two arms were denoted $\texttt{arm}_1$ and $\texttt{arm}_2$. These were the actions available to the agent in state 1. There were no actions available to the agent in state 2 since it was a terminal state. The non-terminal state and action set was therefore

$$\mathcal{S} = \{1\} \quad \text{and} \quad \mathcal{A} = \{\texttt{arm}_1, \texttt{arm}_2\}.$$

This particular 2-AB handed out a reward of $+1$ if $\texttt{arm}_1$ was pulled, and ± 0 if $\texttt{arm}_2$ was pulled. Figure 4.3 shows the transition graph for the 2-AB task. The double border around state 1 marks it as the initial state. This was the state the agent found itself in at the start of every episode. The shaded background of state 2 marks it as the terminal state. Transition arrows are labeled both with the action they represent, and the reward handed to the agent when it selected that action. Note that there are no arrows leaving state 2. The episode ended as soon as the agent entered the terminal state 2.

The 2-AB was chosen because of its simplicity, which allowed for in-depth empirical evaluation and analysis. It was the only environment which did not present the agents with a specific challenge. This allowed it to be used as a performance reference.

The optimal solution was, of course, the arm which gave a reward of one: $\texttt{arm}_1$. The value of an episode u_{ij} was one if the agent selected that arm, and zero otherwise.

Negative Reward Task

This task was based on the 2-AB task. The difference was that the reward for the non-optimal arm was now -1. The optimal arm still gave a reward of $+1$. In other words, the agent was not only rewarded for selecting the optimal arm, but also punished for not

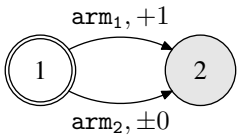


Figure 4.3 Transition graph for the 2-AB task.

selecting it. Such a combination of reward and punishment was expected to increase the speed and stability of learning.

The value of an episode u_{ij} was one if the agent selected $\texttt{arm}_1$, and zero otherwise.

Fuzzy Task

This task was also based on the 2-AB task. The difference was that the reward for the non-optimal arm was now 0.8. Admittedly, this was a very poor definition of a RL problem. Recall that the RL problem is a framing of the problem of learning by interaction to achieve a goal, but in this task both arms yielded high rewards. The optimal arm was worth only slightly more than the other. This meant that the goal was not clearly defined, it was "fuzzy", so to speak, hence the name of the task.

The purpose of the task was to estimate how likely the agent was to get stuck on a suboptimal solution. The reward of 0.8 for the non-optimal arm was chosen a bit arbitrarily. It had to be "high enough", but still significantly lower than the reward for the optimal arm. 0.8 seemed like a reasonable choice.

The value of an episode u_{ij} was one if the agent selected $\texttt{arm}_1$, and zero otherwise.

Relearning Task

This task was in fact composed of two 2-AB tasks. The first 2-AB task was the same task described previously. The second 2-AB task was similar to the first, but the rewards had now switched places with each other. The agent was trained on the first task for 50 episodes. For the remaining episodes it was challenged with the second task. This meant that the agent had to be able to "unlearn" what it had learned during the first 50 episodes. Otherwise it would be unable to find the optimal solution on the second 2-AB task.

The purpose of this task was to investigate the relearning properties of the agents. The ability to relearn is very valuable in a changing environment such as the real world where, at least in the animal kingdom, adaptation is crucial for survival.

The value of an episode u_{ij} was one if the agent selected the arm which resulted in a reward of $+1$ in that episode, and zero otherwise.

Stochastic Task

This task was also based on the 2-AB task. The difference was that rewards were not handed out deterministically. If the agent selected $\texttt{arm}_1$ it received a reward of $+1$ with a probability of 90 %, and a reward of zero otherwise. The second arm, $\texttt{arm}_2$, yielded a reward of $+1$ with a probability of only 10 %, and zero otherwise. Many real-world problems are stochastic by nature in the sense that the optimal solution may not always yield a high reward all the time. The first arm was clearly the optimal solution to this task, and yet it would not always yield a reward. To add to the confusion, the second arm would sometimes yield a reward.

The purpose of this task was to give an indication of the agent's capability to handle stochastic tasks. The probabilities of 90 % and 10 % were chosen a bit arbitrarily. The optimal arm had to yield a reward of $+1$ most of the time, whereas the other arm had to yield a reward of ± 0 most of the time. 90 % and 10 % seemed like reasonable choices.

The value of an episode u_{ij} was one if the agent selected $\texttt{arm}_1$, and zero otherwise.

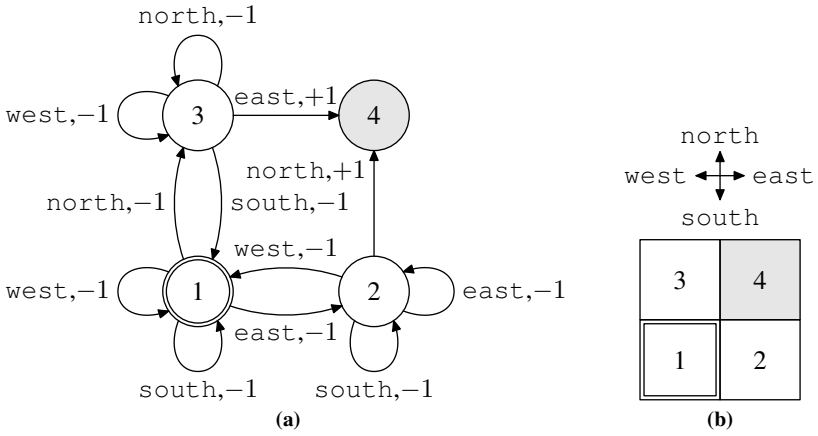


Figure 4.4 Illustrations of the 2x2 GW task. The transition graph is shown in (a), and a more compact illustration of the states and actions is shown in (b).

Delayed Reward Task

The bandit tasks could not be used to evaluate how the agents coped with delayed rewards, i.e. when the consequence of an action is not immediately observable. Most real-world tasks fall into this category. The task used to present this challenge to the agents was a so called 2-by-2 gridworld (2x2 GW). The term gridworld typically refers to tasks which have a grid-like appearance. The transition graph of the particular 2x2 GW used in the evaluations is illustrated in Figure 4.4a. Next to it, in Figure 4.4b, the same gridworld is shown in a more compact illustration, where each cell corresponds to a state.

The initial state, which was labeled 1 in this task, is marked with a double border. There was one terminal state: state 4, which is shaded gray in both the transition graph and the more compact illustration. At the start of the episode, the agent was placed in the initial state 1.

In each state the agent could choose between going north, south, east, or west. The agent's location would be left unchanged if it would "step off" the gridworld. This would happen if, for example, the agent chose to go west in state 1. All transitions resulted in a reward of -1, except the ones which brought the agent into the terminal state 4; those transitions yielded a reward of $+1$.

By handing out negative rewards on all transitions which did not lead into a terminal state, the agent was encouraged to find the terminal state as quickly as possible. There were two such solutions, or paths: $1 \rightarrow 2 \rightarrow 4$ and $1 \rightarrow 3 \rightarrow 4$, both of which were considered optimal. The value of the episode u_{ij} was computed as:

$$u_{ij} = \frac{\text{length of shortest path}}{\text{length of agent's chosen path}} \tag{4.8}$$

In other words, an episode value of one meant the agent found one of the shortest paths through the gridworld. An episode value closer to zero meant the agent took a more "scenic" route before ending up in the terminal state. If the agent failed to find the terminal state within 1024 time steps, the episode was terminated with $u_{ij} = 0$. Such a "time step limit" had to be set to make sure all episodes eventually terminated. Though it was chosen a bit arbitrarily, 1024 time steps was considered to be an appropriate limit.

Extended Negative Reward Task

Because of some surprising results for the A_{R-P} agent on the negative reward task, an extension to it was created. This task was also a bandit, but it had ten arms instead of the usual two. Nine of these arms yielded a reward of -1, while only $\texttt{arm}_1$ yielded a reward of $+1$.

The value of an episode u_{ij} was one if the agent selected $\texttt{arm}_1$, and zero otherwise.

Strictly Positive Delayed Reward Task

Due to some surprising results for the BCPNN agent on the delayed reward task, an extended task was created: the strictly positive delayed reward task. The tasks were almost identical, the only difference was that all transitions which did not bring the agent into the terminal state 4 now yielded a reward of ± 0. This meant that the agent was in no way encouraged by the reward signal to find the shortest path through the gridworld.

The episode value was computed in the same way as in (4.8).

Chapter 5

Results

The results of the evaluations are presented in this chapter. Before the agents could be evaluated and have their performance compared, a reasonable value for the gain parameter G had to be found. It was important to use a value for G which did not favor either of the agents. The results of this search is shown in Section 5.1. The following three sections presents the evaluation results for the three agents, beginning with the Sarsa agent since it was used for benchmarking. Finally, the last section covers the results of some additional evaluations. These extra evaluations were performed due to some unexpected results during the previous ones.

5.1 Finding a Reasonable Gain

Figure 5.1 on the next page illustrates how different values of the gain parameter G affected the worst-case performance $u_{\min}$ of the agents on the 2-AB task. For this evaluation, the agents were allowed to learn for $E = 2000$ episodes. Note that the performance of the Sarsa and A_{R-P} agents were almost identical; their $u_{\min}$ values overlapped almost exactly. For values of $G < 10$ the BCPNN agent outperformed the other agents, but $u_{\min} \approx 1$ for all three agents at $G = 10$. While it is difficult to make out from the plot, the performance of the A_{R-P} and the BCPNN agent then dropped slightly near $G = 30$. Because of this, it was decided that

$$G = 10 \tag{5.1}$$

was a reasonable choice, as it appeared this value did not favor either of the agents.

5.2 Sarsa Agent

The results of the evaluations performed on the Sarsa agent are summarized in Figure 5.2 on page 33. It would seem the Sarsa agent easily solved all the tasks, except perhaps the fuzzy task; more about that below. The parameter settings that were used are shown in Table 5.1 on the next page. These settings were found to be optimal by means of empirical parameter search, see Section 4.4 on page 26. Parameter γ was found to have no effect on the immediate reward tasks.

The fuzzy task proved to be the most difficult task to solve for the Sarsa agent. Decent performance was achievable with parameter setting: $\alpha = 0.01$. Increasing

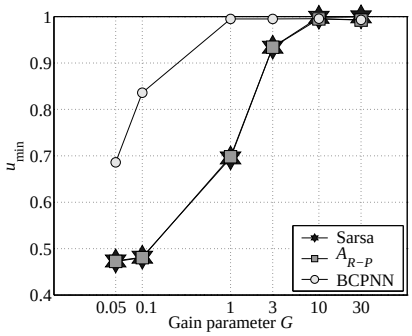


Figure 5.1 The worst-case performance $u_{\min}$ of the three agents as a function of the (log) gain parameter G on the 2-AB task.

the learning rate parameter α too high above 0.01 resulted in a highly unstable performance, with many outliers. Lowering α too much below 0.01 resulted in the agent being unable to learn anything useful within 2000 episodes.

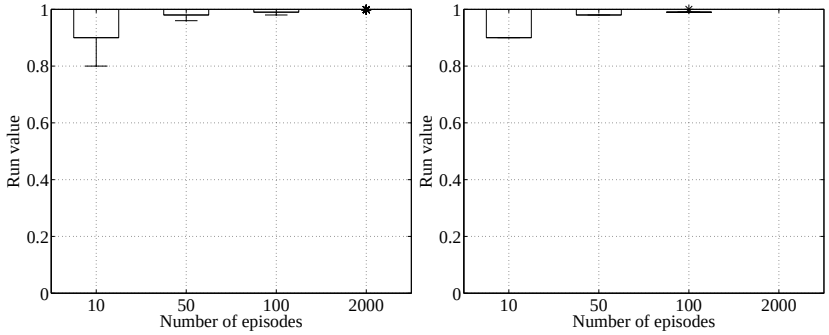
5.3 A_{R-P} Agent

The evaluation results for the A_{R-P} agent are shown in Figure 5.3 on page 35. Overall, the agent performed very well, though it was not very successful on the delayed reward task. The results are discussed in more detail below.

The effect on the worst-case performance $u_{\min}$ with different parameter settings was investigated on the 2-AB task, in accordance with the empirical parameter search method described in Section 4.4 on page 26. The purpose was to learn more about how different parameter values affected the agent's performance, and to find the optimal parameter setting on the 2-AB task. The agent was not allowed to learn for more than 100 episodes in order to get a good spread of $u_{\min}$ values. The result of this parameter search is shown in Table 5.2 on page 34. This data suggested that the optimal parameter setting on the 2-AB task was $\rho = 10^{10}$ and $\lambda \geq 0.1$. It was clear that for higher values of ρ, the other parameter λ had less and less effect. However, note that, even for the extreme choice $\rho = 10^{10}$, performance dropped for $\lambda = 0$. Furthermore, for lower values of ρ, it was clear that the agent performed better as $\lambda \to 1$. Finally, note that the performance of $\rho = 10$ was almost as good as $\rho = 10^{10}$. Because of this, and the

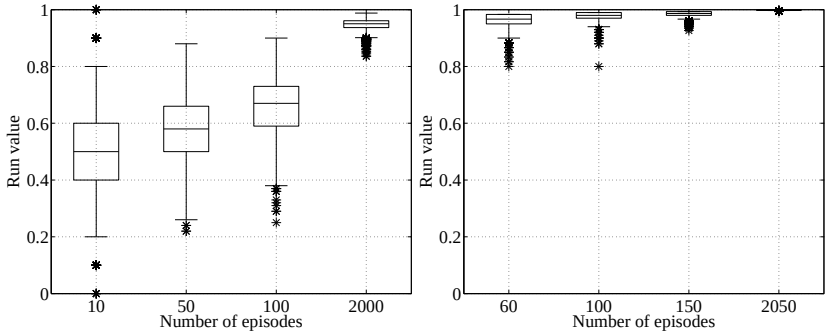
Table 5.1 Optimal parameter settings for the Sarsa agent. Parameter γ had no effect on the immediate reward tasks.

Task	α	γ
2-AB	1	–
Negative reward	1	–
Fuzzy	0.01	–
Relearning	1	–
Stochastic	0.1	–
Delayed reward	1	1



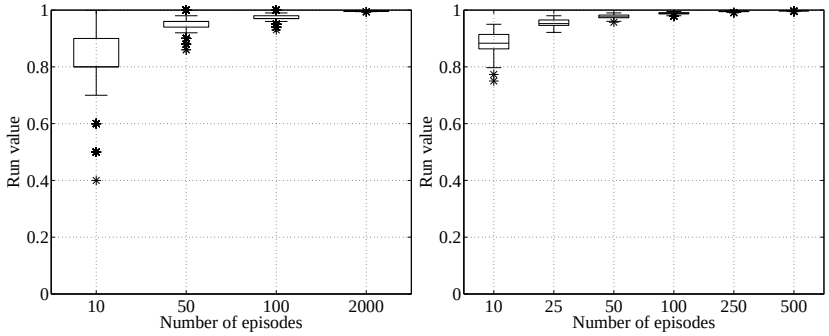
(a) 2-AB task.

(b) Negative reward task



(c) Fuzzy task.

(d) Relearning task.



(e) Stochastic task.

(f) Delayed reward task.

Figure 5.2 Performance of the Sarsa agent on the 2-AB, negative reward, fuzzy, relearning, stochastic, and delayed reward tasks. Each plot shows the distribution of run values as a function of the number of learning episodes. The parameter settings used are shown in Table 5.1 on the preceding page. See Section 4.4 on page 25 for more information about the data displayed in this figure.

Table 5.2 Worst-case performance $u_{\min}$ of the A_{R-P} agent on the 2-AB task with different parameter settings.

	λ				
ρ	0	0.1	0.5	0.9	1
0	0.29	0.34	0.31	0.32	0.32
0.1	0.56	0.61	0.71	0.73	0.74
0.5	0.79	0.81	0.87	0.89	0.88
1	0.85	0.89	0.90	0.91	0.91
2	0.90	0.92	0.91	0.93	0.93
5	0.89	0.93	0.95	0.95	0.96
10	0.89	0.95	0.97	0.98	0.98
10^{10}	0.88	0.98	0.98	0.98	0.98

fact that 10^{10} is such an outrageously large number compared to $\lambda = 1$, the optimal parameter setting on the 2-AB was decided to be

$$\rho = 10 \quad \text{and} \quad \lambda = 1. \tag{5.2}$$

The parameter setting in (5.2) was found to be optimal, by means of empirical parameter search, on almost all tasks. The stochastic task was the exception, where $\rho = 1$ and $\lambda = 0$ was found to be optimal.

It was also of interest to investigate how much the agent could improve its performance when it was allowed to learn for many more episodes. A limit of $E = 2000$ episodes was set, and the agent was evaluated using the same parameter settings shown in Table 5.2. The results revealed that the agent converged to the optimal solution for $\rho >= 0.1$.

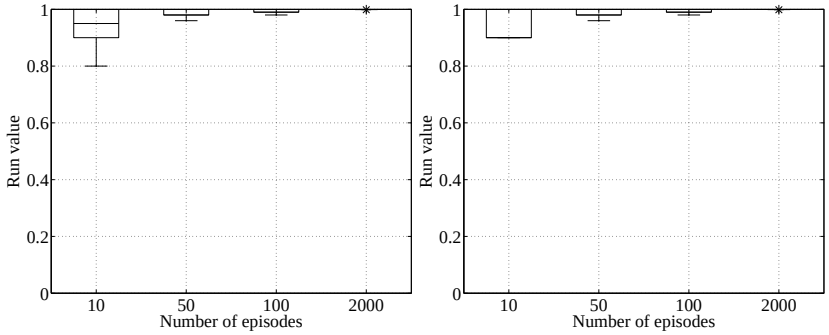
The performance of the A_{R-P} agent on the negative reward task was better than its performance on the 2-AB task. This was unexpected, as the agent was not designed to handle negative rewards, see Section 6.3 on page 42. Because of this, more evaluations were made, see Section 5.5 on page 39.

The agent appeared to converge upon the optimal solution on the fuzzy task, but it was an unstable convergence with many outliers. After 100 episodes, its worst-case performance $u_{\min}$ was still only about 0.7.

Performance on the relearning task was excellent for all values of $\lambda \geq 0.01$ tested. With $\lambda = 0$ performance was terrible. The speed of relearning was found to be faster as $\lambda \to 1$. With the optimal parameter setting in (5.2) the agent was able to relearn faster and with greater stability than the Sarsa agent.

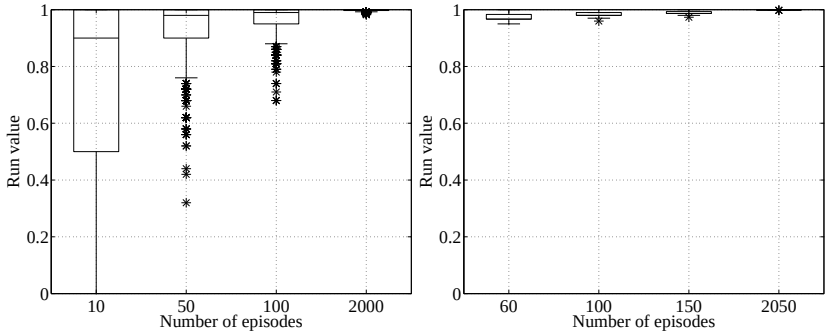
It was difficult to find a parameter setting that could be regarded as optimal on the stochastic task. Different settings were found to have different advantages. Shown in Figure 5.3e on the next page is the performance of the A_{R-P} agent with parameter setting $\rho = 1$ and $\lambda = 0$. With this choice of parameters the agent eventually converged upon the optimal solution. It was a very unstable convergence however, with many outliers. Note that one outlier has achieved a run value only marginally greater than zero after 100 episodes. With the parameter setting shown in (5.2) the convergence was more stable, but it did not converge to the optimal solution within 2000 episodes.

The A_{R-P} agent was not very successful on the delayed reward task. In the limit, as $E \to \infty$, it appears the agent will converge to a performance of about 0.7. While this is better than using a random policy, it is terrible when compared to the Sarsa agent.



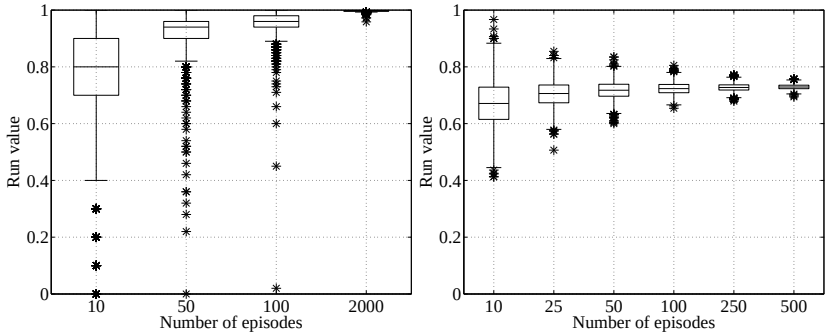
(a) 2-AB task

(b) Negative reward task



(c) Fuzzy task.

(d) Relearning task.



(e) Stochastic task

(f) Delayed reward task.

Figure 5.3 Performance of the A_{R-P} agent on the 2-AB, negative reward, fuzzy, relearning, stochastic, and delayed reward tasks. Each plot shows the distribution of run values as a function of the number of learning episodes. The parameter setting used is shown in (5.2), except for the stochastic task. See Section 4.4 on page 25 for more information about the data displayed in this figure.

Table 5.3 Parameter settings of the BCPNN agent on the tasks shown in Figure 5.4 on the facing page.

Task	τ_E	τ_P
2-AB	1	1
Negative reward	1	1
Fuzzy	25	500
Relearning	5	200
Stochastic	1	1
Delayed reward	5	25

5.4 BCPNN Agent

The evaluation results for the BCPNN agent are shown in Figure 5.4 on the facing page. The parameter settings used are shown in Table 5.3. These settings were all found to be optimal by means of empirical parameter search, described in Section 4.4 on page 26. It can be seen that the agent was not able to solve all tasks in a satisfactory manner. The results are discussed in more detail below.

The effect on the BCPNN agent's worst-case performance $u_{\min}$ with different parameter settings was first investigated on the 2-AB task. In order to get a good spread on the $u_{\min}$ values, learning was limited to 200 episodes. The performance is shown in Table 5.4. If τ_E was set too high, it would completely destroy the learning ability of the agent on the 2-AB task. As an example, compare the effects of $\tau_E = 50$ and $\tau_P = 50$. With $\tau_P = 50$ it is still possible to achieve good performance, but with $\tau_E = 50$ it appears to be impossible. Upon examination of the table, it can be seen that performance was best with $\tau_E = \tau_P = 1$, and thereabouts.

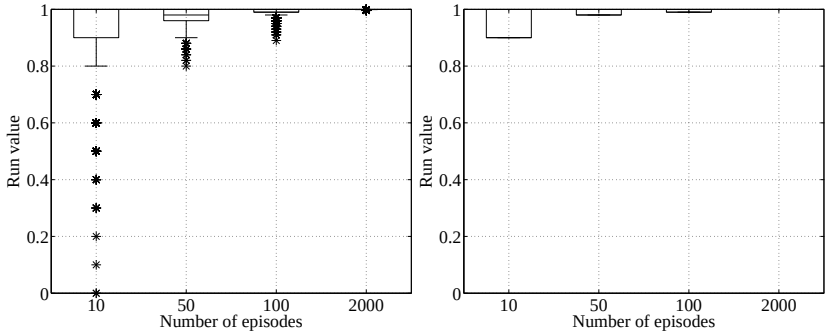
Using the optimal parameter setting, the BCPNN agent eventually converged to the optimal solution on the 2-AB task after 2000 episodes. The convergence was quite unstable though, with many outliers. Even after 100 episodes, the agent's worst-case performance was only about 0.9.

Introducing negative rewards had a positive effect on the stability of convergence. Notice in Figure 5.4 on the facing page how many outliers there are in (a) the 2-AB task, compared to (b) the negative reward task, which shows no outliers at all.

No parameter setting which produced good results on the fuzzy task was found. Some choices of τ_E and τ_P produced a high best-case performance $u_{\max}$, but also a low

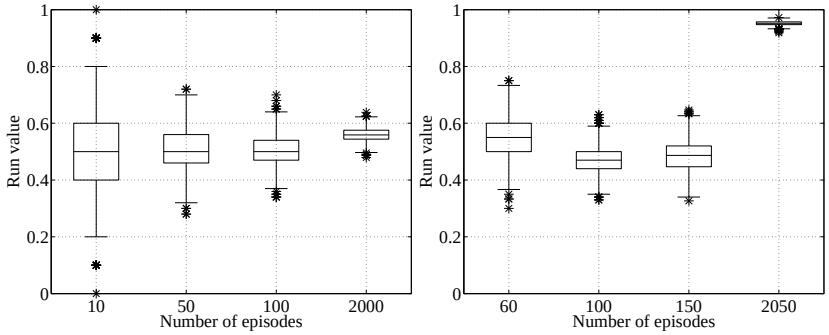
Table 5.4 Worst-case performance $u_{\min}$ of the BCPNN agent with different parameter settings on the 2-AB task.

	τ_P							
τ_E	1	5	10	25	50	100	200	500
1	0.95	0.95	0.95	0.95	0.93	0.88	0.86	0.73
5	0.95	0.95	0.94	0.91	0.89	0.81	0.74	0.54
10	0.93	0.92	0.89	0.86	0.79	0.69	0.57	0.45
25	0.73	0.70	0.69	0.65	0.60	0.53	0.45	0.43
50	0.57	0.55	0.55	0.52	0.51	0.46	0.42	0.40



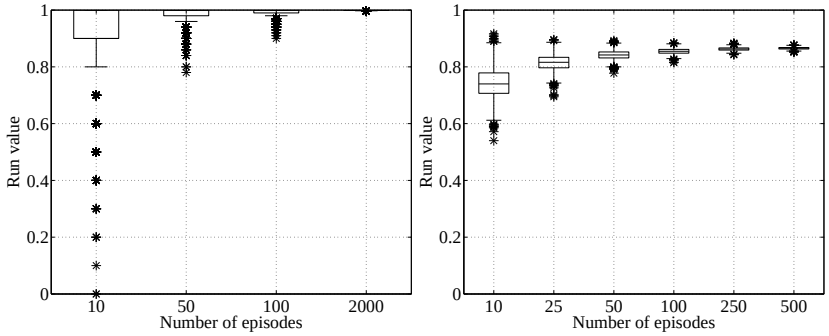
(a) 2-AB task.

(b) Negative reward task.



(c) Fuzzy task.

(d) Relearning task.



(e) Stochastic task.

(f) Delayed reward task.

Figure 5.4 Performance of the BCPNN agent on the 2-AB, negative reward, fuzzy, relearning, stochastic, and delayed reward tasks. Each plot shows the distribution of run values as a function of the number of learning episodes. The parameter settings used are shown in Table 5.3 on the facing page. See Section 4.4 on page 25 for more information about the data displayed in this figure.

Table 5.5 Worst-case performance $u_{\min}$ of the BCPNN agent with different parameter settings on the fuzzy task.

	τ_P								
τ_E	1	5	10	25	50	75	100	300	500
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.12
5	0.00	0.00	0.00	0.00	0.01	0.02	0.03	0.23	0.22
10	0.00	0.00	0.01	0.02	0.05	0.15	0.17	0.33	0.33
25	0.23	0.26	0.24	0.28	0.29	0.32	0.36	0.35	0.39
50	0.37	0.36	0.33	0.36	0.36	0.35	0.36	0.37	0.35
100	0.36	0.36	0.37	0.38	0.37	0.38	0.37	0.37	0.35
300	0.37	0.38	0.39	0.35	0.37	0.38	0.37	0.38	0.36
500	0.36	0.38	0.38	0.38	0.38	0.38	0.37	0.39	0.38

worst-case performance $u_{\min}$ at the same time. Other settings lowered the difference $u_{\max} - u_{\min}$, but resulted in a low $u_{\max}$ instead. For example, with $\tau_E = \tau_P = 1$, the difference was $u_{\max} - u_{\min} = 1$. After 200 episodes of learning, the worst-case performance was still below 0.40 for all parameter settings investigated, see Table 5.5. Figure 5.4c on the preceding page shows the performance with parameter setting $\tau_E = 25, \tau_P = 500$. It was one of the best ones found, and yet it was only slightly better than a completely random policy.

It was not obvious how to set the parameters τ_E and τ_P to achieve good performance on the relearning task, see Table 5.6. In this empirical parameter search, learning was allowed to take place over 2000 episodes. One of the best parameter setting found was $\tau_E = 5$ and $\tau_P = 200$. With $\tau_E = \tau_P = 1$, the agent was completely unable to relearn.

Introducing the challenge of a stochastic environment proved to be of little difficulty for the BCPNN agent. In Figure 5.4 on the preceding page, compare the plot in (a) with the one in (e). The two graphs are almost identical, meaning that performance hardly dropped at all.

The agent was also evaluated on the delayed reward task. Its worst-case performance $u_{\min}$ with different parameter settings, over 100 learning episodes, is shown in Table 5.7 on the next page. It appeared that $\tau_E = 5$ and $\tau_P = 25$ could have been a good parameter setting, but the agent failed to converge upon the optimal solution. This was much worse than expected, and therefore additional evaluations were made, see Section 5.5 on the facing page.

Table 5.6 Worst-case performance $u_{\min}$ of the BCPNN agent with different parameter settings on the relearning task.

	τ_P							
τ_E	1	5	50	100	200	300	400	500
1	0.02	0.02	0.02	0.02	0.55	0.83	0.90	0.91
5	0.02	0.02	0.45	0.88	0.92	0.92	0.92	0.91
10	0.69	0.78	0.89	0.91	0.90	0.88	0.86	0.84
25	0.80	0.79	0.77	0.76	0.73	0.71	0.69	0.67

Table 5.7 Worst-case performance $u_{\min}$ of the BCPNN agent with different parameter settings on the delayed reward task.

	τ_P						
τ_E	1	5	10	25	50	75	100
1	0.36	0.69	0.70	0.71	0.72	0.72	0.72
2	0.45	0.77	0.78	0.80	0.79	0.79	0.78
5	0.43	0.77	0.80	0.82	0.81	0.81	0.79
10	0.42	0.67	0.75	0.76	0.76	0.75	0.74
25	0.39	0.54	0.57	0.60	0.61	0.60	0.60

5.5 Additional Evaluations

Due to some unexpected results for the A_{R-P} and the BCPNN agents, some additional evaluations were made. The results are presented here.

Extended Negative Reward Task

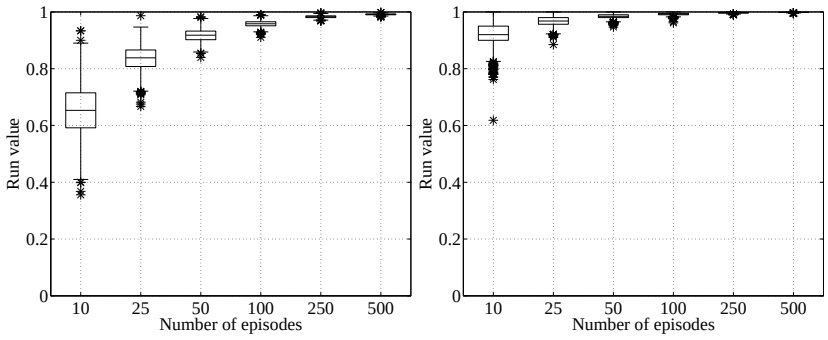
It was surprising to find that the A_{R-P} agent was successful on the negative reward task, because according to Section 3.1 on page 15, the reward r used in (3.2) must be in the interval $[0, 1]$. These results suggested that this may not be entirely true. Therefore, the A_{R-P} agent was evaluated on the extended negative reward task, which is described in Section 4.5 on page 30.

Using the same parameter setting as in the negative reward task, the A_{R-P} agent achieved a worst-case performance $u_{\min} = 0$ after 100 episodes of learning. Several other parameter settings were tried, but no improvement could be found. In other words, the agent was completely unsuccessful. The Sarsa and the BCPNN agents, on the other hand, had no problems solving this task.

One possible explanation for the failure of the A_{R-P} agent, was that it simply was not able to handle bandit tasks with more than two arms. To test this hypothesis, the A_{R-P} agent was evaluated on a bandit with ten arms, where nine of them yielded a reward of ± 0, and one yielded $+1$. The agent was found to be able to cope with this task without any problems. The Sarsa and the BCPNN agents had no problem solving it either.

Strictly Positive Delayed Reward Task

The BCPNN agent performed much worse on the delayed reward task than was expected. In order to investigate this further, a strictly positive delayed reward task was constructed. The task is described in Section 4.5 on page 30. The parameter setting of the BCPNN agent was $\tau_E = 2$ and $\tau_P = 5$. For comparison, the Sarsa agent was also evaluated on this task, with parameter setting $\alpha = 0.1$ and $\gamma = 1$. The results are shown in Figure 5.5 on the following page. It can be seen that the BCPNN agent outperformed the Sarsa agent on this task.



(a) Sarsa agent. **(b)** BCPNN agent.

Figure 5.5 Performance of the Sarsa and the BCPNN agent on the strictly positive delayed reward task. Each plot shows the distribution of run values as a function of the number of learning episodes.

Chapter 6

Discussion

The results of the evaluations presented in the previous chapter will be analyzed and discussed here. Note that this chapter contains, to a large extent, subjective opinions of the author.

The chapter begins with a section covering the results of the investigation of the gain parameter G, and how it affected the performance of the agents. The following three sections discuss the results for each of the three agents separately. The last section makes some overall conclusions regarding the findings presented in this thesis. Some suggestions are made as to how this line of research could be extended.

6.1 Gain Parameter G

The results of the evaluations regarding the gain parameter G, which are presented in Figure 5.1 on page 32, will be discussed in this section. Overall, the data suggests that the gain parameter G can indeed be used to control the trade-off between exploration and exploitation.

Consider what happens as G is increased. For the Sarsa agent, it appears that the worst-case performance $u_{\min}$ of the agent will converge to the optimal solution. This supports the assumption that, at least for this agent, G can be used to control the trade-off between exploration and exploitation. The worst-case performance of both the BCPNN and the A_{R-P} agent, however, has dropped at $G = 30$ compared to $G = 10$. This contradicts the very same assumption. It is, however, a very small anomaly, and no definite conclusion can be drawn. At most, these results indicate that G may have a more complicated impact on the performance of these two agents than initially estimated.

The BCPNN agent achieved a better worst-case performance than the other two agents for low gain values ($G < 10$). This could indicate that the BCPNN agent was able to extract more accurate information about the environment. It is possible that it was able to distinguish the correct solution more clearly than the other two agents. That would explain why it did not need a high gain value ($G \geq 10$) to begin exploiting so heavily.

Interestingly, the data points for the A_{R-P} agent overlapped almost exactly with those of the Sarsa agent, except at $G = 30$. In other words, the worst-case performances of these two agents were almost identical. I have no explanation for this behavior.

6.2 Sarsa Agent

Refer to Figure 5.2 on page 33 while reading the discussion of the Sarsa agent's results in this section.

As was expected, this agent performed very well on the 2-AB, negative reward, relearning, and the delayed reward tasks. The learning rate could be set to its most extreme, $\alpha = 1$, without many signs of instability. The exception being perhaps on the relearning task, where relearning did appear to take place, but with some signs of instability. The agent's performance was increased on the negative reward task, when compared with its performance on the 2-AB task, as was expected. The Sarsa agent also displayed a solid performance on the delayed reward task. For all these tasks the agent appeared to converge to a run value of one. This meant that it would find, and converge to, the optimal solution.

The agent performed worse than expected on the fuzzy task. The agent displayed no signs of converging to the optimal solution after 100 episodes. This, in combination with the many outliers after 2000 episodes makes one wonder what it might converge to as $E \to \infty$, or if it would converge at all. The reason it failed was probably due to the choice of gain $G = 10$, in combination with the choice of rewards for the optimal $(+1)$ and the suboptimal $(+0.8)$ arm.

Due to the nature of the stochastic task, it was quite understandable that the learning parameter α had to be set fairly low to ensure good performance.

6.3 A_{R-P} Agent

This section will discuss the evaluation results for the A_{R-P} agent. It will be helpful to look at Figure 5.3 on page 35 while reading this section.

Looking at Table 5.2 on page 34, it appears that parameter ρ controlled the learning rate, at least up to a point. This was in agreement with the S-model A_{R-P} weight update rule in (3.2) on page 15; the higher the value of ρ, the greater the weight updates will be.

The 2-AB task presented no difficulty for the agent. Its learning speed and stability of convergence matched that of the Sarsa agent almost exactly.

The S-model A_{R-P} weight update rule does not allow $r < 0$, so it was surprising to see that the agent's performance on the negative reward task was so good. In fact, its performance was slightly improved compared to the 2-AB task. This result indicated that the A_{R-P} agent could handle negative rewards after all. However, its terrible performance on the extended negative reward task extinguished that idea. It must be concluded that the agent has, at most, some limited ability to cope with negative rewards.

After having seen the Sarsa agent produce poor results on the fuzzy task, it was unexpected to see the A_{R-P} agent solve it. Convergence was very shaky with many outliers, but in the limit it did appear to converge to the optimal solution.

The performance of the A_{R-P} agent on the relearning task was excellent; in fact, it was even better than the Sarsa agent. The poor results for $\lambda = 0$, combined with the fact that relearning was much faster as $\lambda \to 1$, seems to indicate that λ served to "punish bad behavior". During the first 50 episodes the agent learned to choose $\texttt{arm}_1$, which meant the weight to that action unit was strengthened. Starting with episode 51, $\texttt{arm}_1$ was now the wrong action to choose. The weight which was associated with $\texttt{arm}_1$ should now have been weakened, but it was not with $\lambda = 0$. With $\lambda = 0$, weights can

only be strengthened due to good behavior, never weakened due to bad behavior, as can be understood by examining (3.2). It was also in agreement with the finding that faster relearning took place as $\lambda \to 1$: the higher the value of λ, the greater the punishment for being wrong.

While the agent does appear to converge to the optimal solution on the stochastic task, it was very unstable, especially when compared to the Sarsa agent. It is clear that the A_{R-P} agent has great difficulties dealing with stochastic tasks, at least for $G = 10$. Lower the value of G may boost its performance, since a lower gain would cause the agent to explore more, thereby giving it a chance to discover that there exists a better choice than $\texttt{arm}_2$, namely $\texttt{arm}_1$.

The A_{R-P} agent had no means to handle delayed reward, and it was therefore a bit surprising to find that it was partially successful on the delayed reward task. The reason it achieved some success on this task was probably because the agent learned to favor the actions `north` and `east`. In other words, it most likely became biased to choosing one of those two actions. There is of course no guarantee that such action favoring is of any help to it in more complicated gridworld. The fact that the A_{R-P} agent performed better than expected on the delayed reward task should not be interpreted to mean that it is able to handle delayed rewards.

In summary, the most surprising result was that the agent was so successful on the fuzzy task. The agent also performed much better than expected on the relearning task. As was expected, the evaluations confirmed that the agent is not able to cope with delayed rewards, or with negative rewards. Finally, it would appear that the gain parameter G was set too high at $G = 10$ for the agent to perform well on the stochastic task.

6.4 BCPNN Agent

The evaluation results for the BCPNN agent will be discussed here. Refer to Figure 5.4 on page 37 while reading this section.

The BCPNN agent had a surprisingly shaky convergence on the 2-AB task. Due to the many outliers, it can be concluded that the BCPNN agent was much slower to learn reliably than the other agents on the 2-AB task. This agent clearly required many episodes of training in order to guarantee good performance. The fact that learning was fastest with $\tau_E = \tau_P = 1$ and thereabouts, as seen in Table 5.4 on page 36, was in agreement with (3.6) and (3.7).

Introducing negative rewards with the negative reward task had a big impact on the stability of convergence. Its performance now matched that of the Sarsa agent. Learning speed and stability was expected to increase with the introduction of negative rewards, but these results were even better than expected.

The agent performed poorly on the fuzzy task. It appears the agent would have converged to a run value of approximately 0.5 as $E \to \infty$. Thus, in the agent's opinion, both arms were about equally good, which was not true. This result was quite understandable, since the parameters τ_E and τ_P were set so extraordinarily high. The fact that $u_{\max} - u_{\min} = 1$ for the much more sensible choices $\tau_E = \tau_P = 1$, meant the agent selected one of the arms exclusively in some runs. This was consistent with the results displayed in Figure 5.1 on page 32. Those results indicate that the agent was able to extract more accurate information about the environment than the other agents, resulting in the agent not needing a high gain in order to start exploiting exclusively. So with $G = 10$, a very high gain from the BCPNN agent's point of view, the agent proba-

bly locked immediately on the first arm it happened to select in the first episode. It can be concluded that, at least for $G = 10$, the BCPNN agent can be fooled quite easily by suboptimal solutions. Lowering the gain would most likely solve this problem.

The BCPNN agent needed careful parameter tuning in order to achieve an acceptable distribution of run values after 2000 episodes on the relearning task. With the setting used, $\tau_E = 4$ and $\tau_P = 200$, the parameter τ_P was set so high, it is doubtful the agent had time to learn much in the first 50 episodes before the reward was swapped. Also, there should have been no need to set τ_E to anything but one on this task, since the reward was handed out without delay. Still, it was necessary in order to achieve decent performance within 2000 episodes. Setting $\tau_E > 1$ meant delaying the storage of correlations, and it explains the odd look of the plot. It took time for the agent to register that the reward had moved after 50 episodes, therefore the plot displayed a "dip" before it began to rise.

This implementation was clearly very bad at this relearning task, but why? The reason it failed was, in retrospect, quite understandable. During the first 50 episodes, the agent strengthened the correlation to $\texttt{arm}_1$, as it yielded a reward of $+1$. Starting at episode 51, the correlation to $\texttt{arm}_2$ was now strengthened instead, since that now was the only arm which yielded a reward of $+1$. However, the correlation to $\texttt{arm}_1$ was still as strong as ever, and it remained strong, since the agent had no mechanism to weaken it. Recall that the agent would modify neither projection if the reward was ± 0.

It was interesting to see that the BCPNN agent, apparently, solved the stochastic task just as easily as it solved the 2-AB task. The stochastic nature of the task did not appear to affect the agent at all. It would appear the BCPNN agent has a knack for solving stochastic tasks.

Surprisingly, the agent did not converge upon the optimal solution in the delayed reward task. A possible answer could have been that the trace (E) variables were not able to provide a satisfying short-term memory. This suspicion was laid to rest as the agent solved the strictly positive delayed reward task without any difficulty. In fact, its performance was even better than the Sarsa agent's on that task. The agent found an "optimal" path, even though it, strictly speaking, was not optimal any more. The rewards no longer motivated the agent to find the terminal state of the gridworld as fast as possible, and yet the agent converged upon that solution anyway; as did the Sarsa agent.

To summarize: It proved to be difficult to set the parameters τ_E and τ_P appropriately on several of the tasks. No parameter setting which could produce decent results on the fuzzy task was found. This probably had to do with the gain being set to high at $G = 10$. The reason the agent produced such poor results on the relearning task was most likely due to the fact that this implementation did not decay its projections when the reward was ± 0. Finally, it seems very strange that the BCPNN agent failed to converge to the optimal solution on the delayed reward task, when it was so successful on the simplified delayed reward task. One would have expected the negative rewards used in the delayed reward task to increase learning speed and stability. Recall how the its performance was increased on the negative reward task, compared to the 2-AB task.

6.5 Conclusions and Suggestions for Future Work

The primary goal of this project was to evaluate a selection of biologically plausible RL agents. The agents were to be constructed from biologically plausible models of learning. Four such models were found, of which two were used to construct RL

agents. The evaluations successfully revealed some of their potential strengths and weaknesses. The overall conclusion must be that, based on these evaluations, neither of the two RL agents can be said to be very efficient. However, some useful insights have been made, and the evaluation results have also spawned ideas for future research in this area.

Aside from the evaluation results, another benefit of this project was the development of the RL agent framework presented in Section 4.3 on page 22. While it is very simple, it can hopefully provide a useful insight into how biologically plausible RL agents can be constructed from appropriate models of learning.

The evaluation procedure and performance measure presented in Section 4.4 on page 25 was found to be very useful in assessing the performance of an agent. The list of tasks used to challenge the agents with was by no means exhaustive, but, with the possible exception of the fuzzy task, they all proved to be useful in illuminating different features of the agents.

Next, in the remainder of this final section, some suggestions will be presented on how this line of research could be extended.

Future Evaluations and Analysis

The Sarsa agent was designed to be used for benchmarking, and the fact that it failed to solve the fuzzy task indicates that the task itself was unsuitable for these evaluations. The reason the agent failed may have been that the difference in rewards for the optimal $(+1)$ and the suboptimal $(+0.8)$ arm was not big enough. This ought to be kept in mind if this type of task is used in future evaluations.

It might be illuminating to investigate what happens with the agents' worst-case performances $u_{\min}$ for $G > 30$, see Figure 5.1 on page 32. Specifically, it would be interesting to find out if the performance of the A_{R-P} and the BCPNN agent will continue to drop. Furthermore, some theoretical analysis might reveal why the $u_{\min}$ values for the Sarsa and A_{R-P} agent practically overlap.

The A_{R-P} agent was expected to fail on the negative reward task, and yet it did not. It remains to be seen why this was so. The task is relatively simple, so it ought to be possible to provide an explanation by theoretical analysis.

These evaluations have not, in any detail, investigated how the agents scale on tasks of increasing complexity. For example, one could have the agents solve a 50-armed bandit (50-AB) task, and compare with the results on the 2-AB task. Following the same line of reasoning, one could also have the agents solve a 8x8 GW task, and compare with the results on the delayed reward task. A biologically plausible and efficient RL agent can be expected to display robustness as the complexity is increased.

Future Agents

The effects of the concept of distributed coding has not been investigated at all in this project. It would be interesting to see this area of research extended in that direction.

The memory (P) variable update rules in (3.7) for the dual BCPNN RL system required a positive "print-now" signal κ, and yet the BCPNN agent was capable of solving the negative reward task, as well as the extended one. The ability to cope with negative rewards was due to its use of a negative projection. That concept might potentially be used with other models that also require a positive reinforcement signal. For example, the S-model A_{R-P} weight update rule in (3.2) is only defined to be used with a positive r. It might be possible to extend the A_{R-P} agent by adding a negative

projection, similar to the BCPNN agent, thereby potentially allowing it to cope with negative rewards more effectively.

One possible solution to the problem with the BCPNN agent's poor performance on the relearning task, could be to add a rule which decays both projections if the reward is ± 0. Intuitively this makes sense, since the projections did not help the agent accumulate any reward. Some evaluations were made using an implementation of this approach on the relearning task and the preliminary results were good. An extension to this idea, is to decay the weights of the projections proportionally, with respect to the most recently activated state-action unit pairs. The idea is to focus the decay of state-action weights to the units that were actually responsible for the ± 0 reward, rather than simply decaying all weights.

The A_{R-P} agent's ability to relearn was very impressive. It might be worthwhile to consider incorporating aspects of its weight update rule in future RL agent implementations. Specifically, the term with factor λ in (3.2) looks promising.

The model presented by Doya (see Section 3.3 on page 18), and the water-maze navigation model (see Section 3.4 on page 19) are especially interesting candidates for future work, because they incorporate TD learning. This should make them very efficient in dealing with delayed reward tasks. There might be a problem in that there are two different sets of weights that need to be updated during learning. In the watermaze model for example, one set of weights connects the place cells to the critic unit, and another set connects them to the action units. Also, note that the model presented by Doya is in fact a generalization of the watermaze navigation model. In other words, the watermaze model is one possible implementation of Doya's model.

Based on what has been presented in Chapter 2, incorporation of TD learning methods are, I think, almost a requirement for a biologically plausible and efficient model of learning. TD learning methods appear to capture the essentials of the biological learning process. It makes sense that learning is driven by an error in temporally successive evaluation/prediction errors because learning, I believe, must in some way require energy. It takes energy to induce changes, such as acquiring new knowledge, and nature is not keen on spending energy. Therefore, learning ought to occur only when it is necessary. For example, if an animal accurately predicts its environment, then it is presumably successful in living in it. There will be no need for it to learn anything new. However, if a prediction is wrong – maybe a food source has been moved – then the animal should be encouraged to learn, to adapt, in order to ensure its survival. This is accurately reflected by the methods of TD learning.

Bibliography

Christian Balkenius and Jan Morén. *Computational Models of Classical Conditioning: A Comparative Study* . Technical report, Lund University Cognitive Studies (LUCS), 1998.

Peter L. Bartlett and Jonathan Baxter. *Hebbian Synaptic Modifications in Spiking Neurons that Learn*. Technical report, Australian National University Research School of Information Sciences and Engineering, November 1999.

Andrew G. Barto and P. Anandan. Pattern-recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-15(3):360–375, May/June 1985.

Andrew G. Barto and Michael I. Jordan. Gradient following without back-propagation in layered networks. In *Proceedings of the IEEE First Annual Conference on Neural Networks*, pages II629–II636. IEEE Publishing Services, New York, 1987.

T.V. Bliss and G.L. Collingridge. A synaptic model of memory: Long-term potentiation in the hippocampus. *Nature*, 361:31–39, January 1993.

T.V.P. Bliss and T. Lomo. Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *J. Physiol.*, 232:331–356, 1973.

Matteo Carandini, David J. Heeger, and J. Anthony Movshon. Linearity and normalization in simple cells of the macaque primary visual cortex. *The Journal of Neuroscience*, 17(21):8621–8644, November 1997.

Barry W. Connors and Michael J. Gutnick. Intrinsic firing patterns of diverse neocortical neurons. *Trends in Neurosciences*, 13(3):99–104, March 1990.

Kenji Doya. Complementary roles of basal ganglia and cerebellum in learning and motor control. *Current Opinion in Neurobiology*, 10:732–739, 2000a.

Kenji Doya. Metalearning, neuromodulation, and emotion. *Affective Minds*, pages 101–104, 2000b.

Kenji Doya. Reinforcement learning in continuous time and space. *Neural Computation*, 12(1):215–245, January 2000c.

Kenji Doya. Metalearning and neuromodulation. *Neural Networks*, 15:495–506, 2002.

Kenji Doya. What are the computations of the cerebellum, the basal ganglia, and the cerebral cortex. *Neural Networks*, 12:961–974, 1999.

D.J. Foster, R.G.M. Morris, and Peter Dayan. A model of hippocampally dependent navigation, using the temporal difference learning rule. *Hippocampus*, 10:1–16, 2000.

Ki A. Goosens and Stephen Maren. Long-term potentiation as a substrate for memory: Evidence from studies of amygdaloid plasticity and pavlovian fear conditioning. *Hippocampus*, 12:592–599, 2002.

Kevin Gurney. *An Introduction to Neural Networks*. Routledge, 1997.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2 edition, 1999.

Anders Holst. *The Use of a Bayesian Neural Network Model for Classification Tasks*. PhD thesis, Studies of Artificial Neural Systems, Nada, KTH, September 1997.

Anders Holst and Anders Lansner. *A Bayesian Neural Network Model With Extensions*. Report TRITA-NA-P9325, Studies of Artificial Neural Systems, Nada, KTH, 1993.

Christopher Johansson and Anders Lansner. *An Associative Neural Network Model of Classical Conditioning*. Report TRITA-NA-P0217, Studies of Artificial Neural Systems, Nada, KTH, 2002a.

Christopher Johansson and Anders Lansner. *A Neural Reinforcement Learning System*. Technical Report TRITA-NA-P0215, Studies of Artificial Neural Systems, Nada, KTH, 2002b.

Christopher Johansson, Anders Sandberg, and Anders Lansner. *A Capacity Study of a Bayesian Neural Network with Hypercolumns*. Report TRITA-NA-P0120, Studies of Artificial Neural Systems, Nada, KTH, 2001.

Christopher Johansson, Peter Raicvevic, and Anders Lansner. *Reinforcement Learning Based on a Bayesian Confidence Propagating Neural Network*, April 2003. SAIS-SSLS Joint Workshop, Center for Applied Autonomous Sensor Systems, Örebro, Sweden.

David J. Linden. The return of the spike: Postsynaptic action potentials and the induction of ltp and ltd. *Neuron*, 22:661–666, April 1999. review.

Jeffrey C. Magee and Daniel Johnston. A synaptically controlled, associative signal for hebbian plasticity in hippocampal neurons. *Science*, 275(5297):209–213, January 1997.

Henry Markram, Joachim Lubke, Michael Frotscher, and Bert Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275(5297): 213–215, January 1997.

Pietro Mazzoni, Richard A. Andersen, and Michael I. Jordan. A more biologically plausible learning rule for neural networks. In *Proceedings of the National Academy of Sciences*, volume 88, pages 4433–4437, May 1991.

P. Read Montague, Peter Dayan, and Terrence J. Sejnowski. A framework for mesencephalic dopamine systems based on predictive hebbian learning. *The Journal of Neuroscience*, 16(5):1936–1947, March 1996.

John P. O'Doherty, Peter Dayan, Karl Friston, Hugo Critchley, and Raymond J. Dolan. Temporal difference models and reward-related learning in the human brain. *Neuron*, 28:329–337, April 2003.

Ole Paulsen and Terrence J. Sejnowski. Natural patterns of activity and long-term synaptic plasticity. *Current Opinion in Neurobiology*, 10:172–179, 2000.

Dale Purves, George J. Augustine, David Fitzpatrick, William C. Hall, Anthony-Samuel LaMantia, James O. McNamara, and S. Mark Williams, editors. *Neuroscience*. Sinauer Associates Inc. Publishers, 3 edition, 2004.

A. Sandberg, A. Lansner, K.M. Petersson, and Ö. Ekeberg. *An Incremental Bayesian Learning Rule*. Technical Report TRITA-NA-P9908, Department of Numerical Analysis and Computing Science, KTH, 1999.

Wolfram Schultz, Peter Dayan, and P. Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, March 1997.

Greg J. Stuart and Bert Sakmann. Active propagation of somatic action potentials into neocortical pyramidal cell dendrites. *Nature*, 367:69–72, January 1994.

Richard S. Sutton. Learning to predict by the methods of temporal difference. *Machine Learning*, 3:9–44, 1988.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

Ajit C. Tamhane and Dorothy D. Dunlop. *Statistics and Data Analysis: From Elementary to Intermediate*. Prentice-Hall, 2000.

Gerald Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

Gerard J. Tortora and Sandra Reynolds Grabowski. *Principles of Anatomy and Physiology*. John Wiley & Sons, Inc., 10 edition, 2003. Chapters 12, 14.

Marius Uscher, Jonathan D. Cohen, David Servan-Schreiber, Janusz Rajkowski, and Gary Aston-Jones. The role of locus coeruleus in the regulation of cognitive performance. *Science*, 283:549–554, January 1999.

Niclas Wahlgren and Anders Lansner. Biological evaluation of a hebbian-bayesian learning rule. *Neurocomputing*, 38–40:433–438, 2001.

Florentin Wörgötter and Bernd Porr. Temporal sequence learning, prediction and control: A review of different models and their relation to biological mechanisms. *Neural Computation*, 17:245–319, 2005.