Solving POMDP with Reinforcement Learning and Extended Long Short-Term Memory

Kristian Lerche Nielsen

4th February 2006

Contents

Majdi koja je svijetlost mog dana i radost mog zivota.
Bez tvog plemenitog bica moja bi dusa bila poluprazna.

Og til min mor, for altid at være der for mig og for din kærlighed og støtte. En bedre mor findes ikke.

Chapter 1

Summary of Main Points

This thesis is about integrating Long Short-Term Memory (LSTM) with Reinforcement Learning (RL). This is in order to make an agent capable of solving tasks with temporal dependencies.

The approach is rather novel and is introduced by Bakker in [Bakker, 04]. In this thesis I experiment with Bakkers's RL/LSTM construction and augment it with forget gates, as they are introduced by [Gers,01]. The contributions group into two aspects.

The first aspect is an experimental investigation in the functionality of different key parts of the methodology. The following points will be discussed:

- I reproduce Bakker's results concerning the T-maze with varying corridor length. I will show that Bakker's reported maximal corridor length of 70 can be pushed much further.
- I experiment with the κ-parameter in Advantage-learning and show the large impact of this parameter on the behavior of the agent.
- I show that too few memory blocks has much impact on performance whereas "superfluous" memory blocks do not have a worsening effect.
- I confirm Gers comment in [Gers,01] that the choice of initial gate bias makes little difference.
- I show the clear advantage of using Bakker's novel directed exploration technique compared to undirected, ϵ-greedy exploration.
- I show experimentally that the addition of eligability traces to Advantage learning, called Advantage(λ)-learning has no or even detrimental effect on both ability to learn and convergence time. This is important and surprising as the main reason to introduce eligability traces in the first place is to *improve* convergence time.

In general, I reflect on the combined technique of RL and LSTM along with deeper understanding of the practical aspects.

The second aspect is an integration of forget gates into the RL/LSTM-structure which to my knowledge has not been done before in this setting. I show how these can be used effectively with RL/LSTM and test this construction on a continuous version of Bakker's test problem.

Chapter 2

Preface

This is my master's thesis in computer science at the Department of Computer Science at the University of Copenhagen[1]. It is written under the supervision of Professor Peter Johansen and with the economical aid in form of a scholarship from the "Body and Mind" Research Priority Area under the University of Copenhagen (see [Body and Mind Homepage]).

Any content of this thesis may be used with explicit reference. The code used is public and can be accessed at [DIKU homepage]. Should you decide to print this thesis then note that several pictures and graphs are in color. Although best viewed in color, they are still comprehensible when not.

2.1 What this Thesis is About

This thesis is essentially about a practical implementation of short-term memory. I show and test an agent which has memory of important events in its past. It utilizes this memory in its interaction with the environment.

Before turning to the details of this, I will motivate the approach with a small introduction to standard psychological theory on memory.

2.1.1 Memory from a Psychological Viewpoint

A fascinating example of a control system is the human being or indeed most living creatures on the Earth. The internal state in a human being would be the memory system. In psychology, memory is usually divided into three types: sensoric memory, long-term memory and short-term memory[2].

Sensoric memory is the memory in which the input from the senses are stored in order to be processed by the brain. This could from a computational viewpoint be considered a short-lived "buffer". Either the brain uses the information in this buffer or the buffer is filled with new sensations. While the

[1]www.diku.dk. Datalogisk Institut Københavns Universitet. Universitetsparken 1, DK-2100 Copenhagen East. Denmark.

[2]See any university-level textbook on cognitive psychology. For instance [Gazzaniga, et al. 02].

visual “buffer” is considered fairly short-lived (about a second) the auditory “buffer” can retain information for a long time (up to about 30 sec.).

Long-term memory is loosely “everything we do not forget” and it somehow resides “inside our heads”. An interesting feature is that we do not remember exact “copies” of events but rather “reconstruct” our memory when needed from a few key features.

Short-term memory is the palette of information which we process at the moment. Psychologists consider “memories” to be “fetched” from long-term memory and/or perceptual memory, then “put” into short-term memory where they are processed for a while. This “cache”-like behavior seems to be consistent with human/animal observations.

The exact working of the human memory or even the division into the above three kinds is largely unchartered territory. The “when”, “where” and “how” of memory is almost a mystery when considered at a detailed level. It is difficult to know whether memory is separate from other cognitive processes, like decision making, or plays an integrated role.

While this thesis does not shed any real light on these issues in the human brain it does provide an example of a working system employing a kind of short-term memory.

2.1.2 Thesis Goal

The main focus of this text is short-term memory integrated with Reinforcement Learning.

I describe, implement and test a system, an agent, which acts in a world according to the decisions of an internal decision-making module. This module utilizes a feature very similar to short-term memory. The reason to claim such similarity is because the agent:

1. Recognizes an important event and stores the relevant data from this event.
2. Keeps this information “safe” from being overwritten or altered for a while.
3. Recognizes another important event where it is necessary to “fetch” the information kept in the storage and use the stored information in decision making.
4. Overwrites the content of the storage with new information when necessary. Technically, the agent first deletes the content and then stores new information but the final result is the same.

All this happens in a short time span. It is much more dynamic than a “long term memory”-like structure where information is kept for a long time and where changes are a slow and time consuming process.

This work is mainly based on the phd dissertation of Dr. Bram Bakker, University of Amsterdam, [Bakker, 04]. I reflect on and reproduce some of his results mainly dealing with the first three "enumerations" above, that is the part about remembering. Also, the phd dissertation of Felix Gers [Gers,01] plays an integral part, as well as the original LSTM-text by [Schmidhuber & Hochreiter, 95].

The fourth point about "forgetting" and reusing memory is one of the main points of the thesis. The integration of this with Reinforcement Learning has to my knowledge not been done before and I consider it my contribution to the field.

The approach utilizes neural networks and especially recurrent neural networks (RNN). This thesis, though, is *not* about RNN in general but the specific RNN technique called Long Short-Term Memory (LSTM). Neither is the thesis about short-term memory in general but only the specific memory-like properties which emerge from the LSTM construct I use.

2.1.3 Central Perspectives in the Thesis

The point of the thesis is actually two-fold with a technical perspective and a conceptual perspective. The **technical perspective** is the integration and properties of the RL/LSTM construction. On the surface this perspective is the central theme throughout the text and the rest of the thesis will mainly deal with the obstacles and possibilities of this.

Underneath the technical one is the **conceptual perspective** which is to "remove work or responsibilities from the designer". This is the motivation for the entire technical approach. Machine learning is often an approach in which the human designer does most of the work and the 'tiny AI' is nursed through its existence. Two main problems are present, here demonstrated with two examples.

Example 1: The Labelling Problem

In supervised learning the acting system, henceforth agent, is shown a series of samples together with a series of labels (correct responses) to these samples. Ideally, the agent will learn the connection between samples and labels so that it can respond correctly to future samples which have no labels.

This learning task can be arbitrarily difficult to solve and a central issue is the amount of training data. But for every training sample an external teacher is needed to label the sample correctly. If thousands of samples need labeling this task alone may hinder an otherwise successful application. A solution to this problem would be to have an agent which could "devise" its own labels in some way and thereby relieving the designer form this chore.

Example 2: The Resetting Problem

Most machine learning systems have a tendency to build up noise over time: edge weights diverge or internal states clutter with useless information. The task here is to reset the agent so that it can learn or solve a new, but similar situation. This is almost always done implicitly by the termination of the program and the restarting with new or similar parameters.

The problem is that it imposes two responsibilities on the designer: one is to reset the system which might involve setting several states to a “good” value (often 0). This is the *how* of the system reset. Another responsibility is to define and realize the point of task completion. This is the *when* of the system reset. The “when” might prove more difficult than the “how” as the reset usually can be performed in the same way whereas the point of task completion involves interaction with the environment.

The designer needs to do both of these things for the agent. The most important of these, however, is where to reset. Of course, it would be nice if the agent could learn how to reset as well. This, however, is typically not too great a task to put on the designer as the same kind of reset can usually be employed throughout an agent’s lifetime.

While these aspects are acceptable and often necessary in laboratory settings they will always be a burden to the designer. As long as the he needs to address all these issues the techniques will have trouble rising to a higher level.

If both the label problem and reset problem are addressed we achieve two substantial benefits. For one we may be better able to design a system for an environment we know nothing about. If it can cope with the mentioned problems, it will be more suited to solve the tasks that we know little about beforehand. Another benefit is that we may let the agent handle more complicated subtasks while leaving us the possibility of acting at a higher level of abstraction.

I summarize the essence in this corny, yet precise, quotation:

> Ask not what you can do for your robot;
> Ask what your robot can do for you.

While this thesis do not completely eliminate the labelling problem nor the resetting problem but I will show approaches which take substantial steps in the right direction.

2.1.4 Structure of the Thesis

I begin the description with an introduction of the type of problems, MDPs and POMDPs, which are relevant in this setting. Also I describe a concrete test problem, the T-maze, which plays a main role in the text. The main part of the thesis is divided in two areas: theory and experiments.

Theory

The theoretic part begins with a description of certain aspects of RL which are beyond the elementary level. Primarily, I will describe Advantage(λ)-learning as it is introduced by Bakker [Bakker, 02] and used in his dissertation [Bakker, 04]. Also, I consider directed exploration and its relevance to POMDP. This account will build on rudimentary knowledge of RL, as is summarized in appendix B.

This is followed by two chapters devoted to the memory aspect of the system. In the first chapter, I motivate the approach with an account of **Recurrent Neural Networks (RNN)** and their drawbacks. The second chapter covers the design and algorithmic details of the **Long Short Term Memory (LSTM)** technology. I consider this section difficult on a first reading and have made an effort to make it as accessible as possible. Among other things, I have tried to include details which are usually omitted in the references.

Experiments

The experimental part is also divided into two. In the first chapter I present Bakker's results. Besides reproducing these I also make various experimental comparisons and reflections on several selected key features.

Bakker's work is mainly based on the original LSTM design as he does not really utilize the developments by Gers. My goal in the second chapter is to integrate Gers' work with RL. The agent is augmented with the ability to forget and re-memorize information. I describe and test this approach which I consider the main contribution in this thesis.

2.1.5 Prerequisites of the Reader

I assume the reader is familiar with **Reinforcement Learning (RL)** on a level similar to the classical book [Sutton & Barto,98]. Also, a general understanding of Neural Network (**NN**) fundamentals (including the backpropagation algorithm) is also assumed. I have found Sutton and Barto's work indispensable for RL and [Mehrotra et al, 96] a satisfactory choice for NN.

In appendix B and C I briefly remind of RL and NN basics, respectively. These are intended to reacquaint the reader with the fields and not as an introduction for the novice.

Lastly, a good grasp on elementary statistics and probability is recommended.

2.1.6 Previous Work

The development of the part of the field relevant to this thesis is roughly as follows.

- 1986 Mclelland and Rumelhart [McLelland, 86] introduces the multilayer feed-forward neural network along with the backpropagation algorithm. Also, they introduce the recurrent neural network and an adaption to back propagation.

- 1987 and 1988 Willams and Zipser develop “Back-Propagation Through Time” (BPTT) [Williams & Zipser] based on Mclelland’s & Rumelharts’ work. Robinson and Fallside develop “Real-Time Recurrent Learning” (RTRL) [Robinson & Fallside]. Both are backpropagation algorithms for RNNs.
- 1991 Hochreiter shows that the back propagated error signal in both BPTT and RTRL tends to blow up or vanish in an exponential manner.
- 1997 Schmidhuber and Hochreiter introduce LSTM as a remedy to the problems with RNN. They show extremely promising results with time-series data [Schmidhuber & Hochreiter, 95]. .
- 2001 Gers improves on the LSTM design with among other things the ability to “forget”.
- 2004 Bakker integrates LSTM with RL and thus uses this otherwise supervised learning technique in a unsupervised learning setting.

In 2005 Hansen and I [Nielsen & Hansen, 05] worked on RL on large or continuous state spaces. We worked with MDP-problems with many similarities to the POMDP-problems in this thesis and, to some extend, this thesis is a further development on this work. They complement each other in the sense that the approaches in this thesis can be carried into more the complicated domains with the techniques described in [Nielsen & Hansen, 05].

2.2 The Value of Implementation

This work is based on a 6-month full time implementation effort. Such work should not be considered a mere “implementation” but rather a “software development process” complete with analysis, design, test, re-design, re-implementation and so forth.

I strongly believe in implementation as part of a learning process. One seldom has the theory completely right if one cannot account for the tiny details which an implementation requires. Personally, I have found and straightened significant misconceptions in my own understanding during this process. These were not discovered even after several readings of the material.

In this aspect I must express my gratitude to Dr. Bakker who has kindly supplied me with his own code from his dissertation. While I have not used any of his code directly in my own applications it has been of great importance to have a working implementation at hand.

By comparison of my own program’s execution with Dr. Bakker’s, I have been able to uncover some extremely hard-found problems in my application. Those would not have been spotted during normal testing, as their impact on convergence time and success rate are hidden. This, I believe, has saved me at least a month of development time.

My final program never agreed completely with Dr. Bakker's program. As my application passed my tests, I decided not to pursue the matter further and assume the difference lies in slightly different initialization procedures. The results in the experimental section are all made after the development reached this point.

I have published my own code at DIKU's website ([DIKU homepage]) but I also suggest a look at Felix Gers' on-line LSTM package written in C. This package is the recommended code by Schmidhuber and can be accessed on his website [New AI].

The bottom line of this is that special care should be taken if one seeks to achieve anything with LSTM-methods. During my career, academical as well as business wise, I have seen many failed projects which mainly shipwrecked on implementation related difficulties. To approach LSTM and RL without delicate attention is to ask for trouble.

That said, the possibilities with LSTM and RL/LSTM are immense, as is shown in this thesis along with the main contributors of the field: [Schmidhuber & Hochreiter, 95], [Gers,01] and [Bakker, 04].

2.3 The Paragraph Noone ever Reads

On the professional level I would like to thank my supervisor Professor Peter Johansen for valuable feed-back throughout this process. I would also like to thank the "Krop og Bevidsthed satsningsområde" ("Body and Mind" see [Body and Mind Homepage]) for sponsoring this project.

As mentioned, I wish to show my greatest appreciation of the help I received from Dr. Bram Bakker. Not only with the implementational issues mentioned above but also with reflections on the techniques in personal communications.

I would like to thank Carsten Birck Jensen and Emil Soelberg Frisendal for thorough proofreading and general help in making the material accessible to other people than the author.

On the private level I would like to thank my fiancee, Majda Spahovic, for support and understanding for these past 8 months, especially the last few. In general, I wish to thank my family and friends for putting up with the lag of time I have had to spend on them.

Chapter 3

RL/LSTM-problemtypes

This chapter is concerned with the type of problems which we wish to solve.

In the first section I informally describe MDPs and POMDPs which are general classes of problems. RL is suited to solve MDP problems but is not able to solve POMDP unaided. The second class, POMDP, is the focus of this thesis.

The second section is about the specific test problem which I use throughout the thesis. It is a simple, small and finite POMDP called the **T-maze** and it is easy to understand and analyze.

3.1 MDP and POMDP

The MDP-field and POMDP-field are rather large and have connections to control theory, optimization theory, statistics and others. They have precise definitions and structure but instead of a rigorous coverage I will describe them in a more informal manner.

A **decision process** is a series of states in each of which an **agent** has to make a decision. This agent can be anything from a casino-player to a nuclear plant operator to a robot. In each decision at state t the agent chooses an action which leads to a new state at time $t + 1$. The new state, s_{t+1}, might be the same as the old, s_t, but the time stamp will always be different.

An important point is that the agent has *influence* on what state it will arrive at next (not complete control, as the influence the action has on the environment is not certain).

This is contrary to regular stochastic processes, like for instance Markov chains. Here we do not talk about an acting agent but rather a manifestation of some complicated phenomena in the world. For instance, in rolling a dice several times the total sum of "eyes" would be a stochastic process. This would actually be a Markov chain but no acting entity is involved.

3.1.1 MDPs, Memoryless Decision Making

A **Markov Decision Process (MDP)** is a series of states in which an agent needs to make a decision based on the current state.

What is important is that an optimal decision can be made solely on the basis of the current state. Whether the agent makes the optimal decision or not is not important. If it is *possible* to make an optimal decision the agent (usually) is able to learn it. I will clarify with a few examples.

MDP example 1: Backgammon

The game of backgammon is my prime example of an MDP. We can decide our move solely based on the position. Also, the “emotional state” of the opponent plays a lesser role than for instance in chess.

More interestingly, though, there is a precise stochastic element. If it is your turn (and you have not yet rolled the dice) you can reason about all your possible moves *combined* with the probability for having the chance to make them. While we usually just throw the dices and think afterwards this is an important aspect when reasoning about the consequences of our actions: “How can my opponent counter this move? What is the probability that he can do that?”.

If the same position is encountered in the same game or in some other game, then the best move is the same. The past game history is not needed. For example, two players can leave a game and two others can take over without knowing how the game evolved to this state and without sharing information with the first players.

Notice that even though the optimal way to play Backgammon is a hard problem, the state space is about 10^{20}, the decision is still only based on the current state.

MDP example 2: Checkers and chess

In checkers the best move can also be determined solely on the basis of the current position of the pieces. Here we have no stochastic element, except perhaps the actions of the opponent. This does not change the MDP-property, though.

Chess is almost, but not quite, an MDP. On the face of it, an agent can make an optimal decision in every situation solely based on the board position. In some situations, however, history plays a role. For instance, whether the king has moved or not has an impact on the possibility of ’castling’.

Psychology may also play a larger role here than in the other two games. As the optimal move in every situation is an open question, players consider the entire history of the opponent’s moves in order to help them make a decision. If one disallows ’castling’ and a selected few other rules, then chess would in theory be an MDP. Though, an MDP which no one currently is able to solve to optimality.

MDP example 3: Heating-control systems
For a non-game example let us consider a controller for a heating system in a brewery where a tank (of lager etc.) has to be kept at a constant temperature. At each state (every 5 minutes or so) a controller can read the current temperature and this alone is enough to decide how much the heating system needs to be adjusted. We here assume that the liquid has stabilized its temperature after the 5 minutes interval so that the state the controller reads is indeed relevant.

3.1.2 POMDPs, including Memory

The **Partial Observable Markov Decision Process (POMDP)** almost needs an abbreviation for the abbreviation. Basically it is a decision process in which the decisions are MDPs but where the decision maker does not know this. The full state is simply *partially observable* to the agent and the unobservable parts play a key role in the environment dynamics.

This seemingly self-contradictionary description can be clarified with a few examples.

POMDP example 1: Whist
The card game whist is a "light" version of the game Bridge. In Denmark, it is probably as popular a game as chess but it should actually not be considered an MDP and not even "almost" MDP like chess.

Just before the game begins several important decisions are made: Which player will "lead" the game (or have the contract)? Which ace-card will be his "partner-ace"? What suit will be the trump suit? Which kind of contract is made? This information is relevant for optimal play all the way to the end.

During the game many other aspects are disclosed. These are specific aspects like who is partner or who is "renonce" but actually all played cards and who played them is information worth considering, especially in the final part of the game.

At a given point in the game it is not possible to play optimally or even correctly if one forgets which suit is the trump suit. This information is not present during any state of the game, it was only "present" in the pre-game state.

If one wishes to truly play optimally the entire history of all played cards will be needed. While such a feat of memory is usually not accomplished it is still necessary to keep track of several important events during the game to be a decent player.

Like chess, another way to view this is that a new player joining a game of whist *cannot* participate meaningfully unless he is told a great deal of background information of the current state of the game. In order to play optimally the newcomer must be given the entire game history.

Once all this information is remembered, though, the game is "reduced" to an MDP but in which each decision state is the currently played cards on the table (current sensor input) *and* the internal state of the player.

If one could “look” inside the heads of the players one would find all the information needed of this game to become an MDP and since we cannot do that we consider it “partially observable” information.

POMDP example 2: Heating-control system
With such an abundant number of real-world examples I still choose to consider our brewery-heater from above. Imagine that two different kinds of ale can be in the tank, needing different temperatures. The temperature regulation problem will now always depend on which kind of beer is in the tank. Once this is stored in an internal state of the controller, the task becomes an MDP.

Notice that almost any decision process can be considered a POMDP. I choose these examples to emphasize that the most “useful” POMDPs are the ones that depend on a fairly limited amount of history (or non-perceivable) information. Also, I assume that the agent has been exposed to the relevant information at some point during its existence.

The decision to buy or sell a certain stock might well depend on information such as the entire history of the stock market and the current (and past) content of all newspapers/news broadcasts. The hidden information becomes so large that it is very difficult to imagine a solution. In this thesis I generally consider POMDP-problems as those where a good or even optimal solution is realistic.

3.2 The T-Maze, a Classic Problem

In this section I describe the test problem I use in the greater part of this thesis.

3.2.1 Maze Structure

Figure 3.1 shows the classical T-maze problem. The agent starts in the far south end of the corridor and can attempt to move one step north, east, south and west. In the corridor the agent may move north or south, east- and west moves will not result in state change.

At the north end there is a junction where the agent can move either east or west. One of the directions is the correct one and if the agent moves in this direction, it is said to **win** the maze and the episode terminates. If it moves in the wrong direction the episode terminates as well and the agent is said to have **lost**. When an agent either wins or looses, ie. performing an east or west action at the junction, it is said to **escape** the maze. The agent has to make a decision at each time step and has to perform an action.

If the agent tries to move into a wall, it receives a small negative punishment, -0.1, and is not moved. If it loses, it also receive a punishment of -0.1 and is restarted at the beginning. If it escapes, it is restarted at the beginning and receives a reward of 4.0. All other actions result in zero reward.

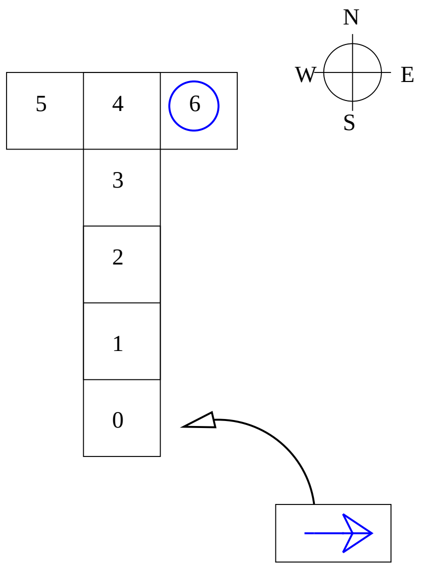

Figure 3.1: The classic T-maze. The agent starts out in state 0 and may move N,E,S or W unless it is obstructed by a wall. The goal is randomly placed in either state 5 or 6 at the beginning of an episode. The placement of this goal is indicated by a “road sign”, the arrow, in state 0.

If the escaping corridor is fixed, say east, the problem is straightforward and conventional RL solves it easily and quickly. Even in a continuous state space the problem is still trivial for RL with function approximation as Hansen and I showed in [Nielsen & Hansen, 05].

By letting the escaping corridor be selected at random (0.5 probability) at the beginning of each episode, we make the problem unsolvable by RL as it cannot be determined where the winning corridor is. A crucial point is to disambiguate the ambiguous junction in which the agent is unable to make an optimal decision.

One approach is to put a **road sign** in the junction, showing which way the winning choice is. This is the basic setup which we worked with in [Nielsen & Hansen, 05] but instead of the discrete case we looked at continuous state spaces and also more complicated mazes. One main result was that RL can solve this problem, even in very complicated mazes, as long as the ambiguous situations (junctions) are made unambiguous with road signs.

In this thesis I move the road sign south to the starting position. Now the agent will have to "remember" the value of the sign all the way up to the junction where it will have to use it in order to make a correct decision. If it forgets the sign, it will not be able to choose correctly at the junction.

It only sees the road sign once, namely as it starts (unless it moves back to the starting position), and has to store the information correctly. In the corridor it has no use of the road sign and the "memory" should therefore not interfere with the agent's actions. Similarly, the corridor sensations should not tamper with the information in the memory.

In the junction the information has to be released in order to make a correct decision. Remember, the agent has "only one chance" as the decision has to be made immediately. As it cannot "sit and contemplate" the situation for a while, the memory has to be released in an immediate fashion.

At the beginning the first state is encoded as either 101 (goal is to the west) or 011 (goal is to the east). The corridor is encoded as 110 and the junction as 001. The two hallways need no encoding as the episode terminates immediately. For updating purposes, the starting state of the next episode is used as next state. The length of the corridor may vary significantly, see the chapter on experiments.

From the agent's point of view, the maze would look like figure 3.2.

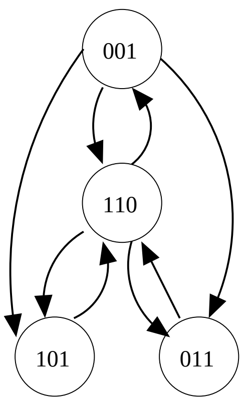

Figure 3.2: When viewing the world from the point of the agent this state diagram will be the result. Transition arrows are added from the junction state and back to the two starting states. This is the way the agent will perceive the episode termination, especially with regard to updates.

3.2.2 Why the T-maze?

To focus on just one problem may seem limiting but it involves many key points of any POMDP problem:

1. It has ambiguous states.
2. It has unambiguous states (which actually are a majority of all the states).
3. It has temporal dependencies.
4. It can be solved sub-optimally without using the temporal dependencies but can solved optimally only by using these.
5. The temporal relations can be of varying length back in time.

I use it as a working example in order to make the theoretical discussions more concrete. The results obtained in the experimental parts on this problem will carry to the more general setting as well. I refer to it as the **T-maze problem** or **our T-maze problem**.

It is also one main problem from Bakker's dissertation which I wish to analyze. I later change the problem somewhat when I augment the agent with an ability to forget.

3.3 Other Approaches to Artificial Cognition

The field of machine learning (**ML**) or artificial intelligence (**AI**) is to broad to cover her in much depth. When considering any of these other approaches it is necessary to focus on their abilities with regard to the T-maze problem, or to MDPs and POMDPs in general.

A large subfield of AI or ML is the supervised learning techniques and the dismissal of these is one of the main points of this thesis. Examples of these are neural networks, recurrent or not, and Support Vector Machines. For all their impressive results in for instance pattern recognition, their main drawback is that they need a “guide” or “teacher” to explain what they should do or not do. This has already been touched on in the preface and we may therefore consider techniques like neural networks to be less suited for MDPs and consequently for POMDPs and the T-maze problem.

Another subfield of ML, not quite so large, is the unsupervised learning techniques. Here, one hears terms like the EM algorithm and others. An essential element is that the structures are self-organizing to some extent, leaving out the problematic teacher.

The problem here is if the teacher is left out completely. Some teacher input might boost the performance considerably as the teacher can point the agent in the right direction. Otherwise, it might take the agent a long time to realize that moving into walls is not productive. In POMDPs it might prove too complex to learn temporal relations if *no* direction at all is presented to the agent.

Too little or too much “teacher intervention” has pros and cons and RL seeks to take the best of both. In RL, the only place an external teacher needs to interact with agent is when defining a reward function which governs the good and the bad actions in the environment. In its extreme, a reward function is similar to supervised learning but this is usually not needed. Rather, a few global events (hitting a wall, reaching the goal) need have a reward defined and the rest might be zero. Over time an agent learns to deduce these rewards.

RL is by itself a large field with numerous different techniques and methods ranging from dynamic programming to temporal difference learning. The specific parts I use are discussed later.

In summary, both pure supervised and pure unsupervised techniques are less suited to handle the specific kinds of problems we are interested in, namely MDPs and POMDPs. Supervised techniques need to label too much which is a crucial bottleneck. Unsupervised techniques seek to find their own “structure of the world” without utilizing perhaps obvious aspects of the environment. The structure in the problem is used to a high degree in RL through the use of the reward function and the value functions. This makes RL ideal to handle these problems.

3.4 Summary and Reflections

In this chapter I have described the main problem types, MDPs and POMDPs, which is the focus of this text.

RL is designed to solve MDP and is therefore not equipped to solve POMDP as it cannot “remember” the information from previous states. By integrating a memory structure into RL it may be possible to overcome this and begin to explore the much richer problem domain of the POMDP problems.

I have also introduced my primary test problem, the T-maze, which I use in the main parts of this thesis. I use this as a “running example” to facilitate the understanding of the theoretical details.

The problem is rather simple but has many essential features which are prominent for any POMDP problem, especially finite-state ones. So the extensive testing I do in the experimental section will carry their results to a more general setting and will help practitioners understand their concrete, real-life situation better.

I do, however, modify the problem later in order to reflect on other parts of the technique. Especially, forgetting is not necessary to solve this problem.

Part I

Theory

Chapter 4

Reinforcement Learning, Selected Topics

In this chapter I focus on some aspects of RL which are especially relevant for our T-maze problem and for POMDPs in general.

First, I describe eligability traces and Advantage learning which leads to Advantage(λ)-learning, a recent development by Bakker [Bakker, 02].

The next topic is models, model-free learning and exploration. This leads to Bakker's novel directed exploration technique which is designed specifically to POMDPs solved with RL.

Lastly, I will reflect on these issues when it comes to POMDP and the T-maze problem.

4.1 Advantage(λ) Learning

This and the following section is based on rudimentary RL knowledge, as is summarized in appendix B. The reader might wish to consult this before reading further.

I use the action value function, $Q(s, a)$, as an initial example but will later move to the Advantage value function, $A(s, a)$. Advantage learning (update of the Advantage value function) is introduced in [Harmon & Baird, 96] and Bakker builds on this to produce Advantage(λ) Learning in [Bakker, 02] and [Bakker, 04].

4.1.1 On- and Off-policy Learning

A well-known RL algorithm is **SARSA**. SARSA is a technical abbreviation of "State, Action, Reward, next State, next Action".

A given state/action-pair at time t, $Q(s_t, a_t)$, is updated by the immediate future reward, r_{t+1} and Q-value of the future state, $Q(s_{t+1}, a_{t+1})$.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \lambda Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{4.1}$$

where α is a real-valued learning rate and $\lambda \in [0, 1]$ is the discounting factor.

The part inside the square brackets is the **error**. The general goal of all RL algorithms is to minimize this error for all state/action-pairs. Over time, the agent should sample enough rewards so that the estimate in $Q(s_t, a_t)$ becomes close to the true value.

The approach to do this may differ, as evident in the equally well-known algorithm **Q-learning** which is updated by

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \lambda \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{4.2}$$

The subtle differences between (4.1) and (4.2) lie in the use of the max operator. This difference is intimately related to the *policy* which is used to update the Q-values. Some explanation is due.

Notice in, for instance SARSA update, that the update is based on the current Q-value and the Q-value for the next state (so the update is technically not made until we know this state). If the reward, r_{t+1}, was absent the system would repeatedly update its action value function based solely on this value function. Perhaps this would converge to a fixpoint, perhaps not.

What makes this self-feeding update useful is introduction of the reward, r_{t+1}, which should be considered a sample from the reward distribution which is unknown to the agent. These rewards are the key to construct a useful value function.

How the agent samples rewards is based on the policy it uses to make its actions. I refer to this as the **acting policy**. This policy might be closely tied to the value function or might be completely independent from this. For example, a policy selecting any action with equal probability would be independent from any value function. Generally, however, the best results are achieved if the policy uses the value function, at least to some extent, in its decisions.

Notice in SARSA how the next value, $Q(s_{t+1}, a_{t+1})$, is *the one chosen by the acting policy* and the total update of the action value function is therefore based on

1. The rewards as they are sampled by the acting policy
2. The value we use in the next state is chosen by the acting policy.

This is called **on-policy update**.

Q-learning is different, though. The reward is still sampled according to the acting policy but the next action value is *not*. Instead it is chosen among all possible next state/action-pairs to be the pair with the maximum value. We say that we use the **optimal policy** to select the value from the next state which we use in the update. As the optimal policy is different from the acting policy this is called **off-policy updates**.

While this distinction may seem strange it makes a difference when considering model-free learning as explained later. If the policy uses on-policy update it will (to some extent) update the Q-table no matter what it encounters. If it has found a good path to the goal but starts making many suboptimal exploring actions, then these exploring actions will update otherwise "good" Q-values with samples from more or less useless parts of the environment.

Perhaps we would wish to "go exploring" in less promising parts of the state space without this having too much effect on our Q-values. Q-learning do this by having one policy to guide the agent, usually an exploring one, but keeps updating the Q-values based on the best experiences so far.

Q-learning, or off-policy learning, is more appropriate in my problem. As ambiguous states will lead to much "confusion" it is better to update a policy based on the best known values so far than to change values according to these many changing rewards.

A more thorough coverage of on- and off-policy updating can be found in [Sutton & Barto,98]. Now, I will turn to a further development of Q-learning which is called Advantage learning.

4.1.2 Advantage Learning, improved Q-learning

As generally known, state spaces need to be fairly small in order for RL to work properly. Larger spaces, and continuous ones, demand function approximation of some kind to represent the value function[1].

Advantage learning was conceived by [Harmon & Baird, 96] in an attempt to make the Q-learning algorithm more suitable to continuous domains. In continuous time RL the values of adjacent states differ by small amounts, relative to the overall variance. Small differences can easily be lost in noise.

Advantage learning seeks to emphasize the difference between action values so the noise will have lesser effect. Even in noiseless environments Advantage learning may still be preferable, especially when employing function approximations which are kind of "noisy" in their own way.

The Advantage value of a state is defined as

$$\max_a A^*(s,a) = V^*(s)$$

where $V^*(s)$ is the optimal state value in state s (see appendix B). The optimal advantage for all states and actions is defined as

$$\begin{aligned} A^*(s,a) &= V^*(s) + \frac{E(R_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a) - V^*(s)}{\kappa} \\ &= \max_a A^*(s,a) + \frac{E(R_{t+1} + \gamma \max_a A^*(s_{t+1},a) | s_t = s, a_t = a) - V^*(s_t)}{\kappa} \end{aligned}$$

Which is the Bellman optimality equation for the Advantage value function. Here, E is the expectation of the reward of taking action a in state s. $\gamma \in$

[1]For a more thorough coverage of this problem I refer to [Sutton & Barto,98] and my previous work ([Nielsen & Hansen, 05]) in which we cover many areas. Most other references use just single techniques.

$[0, 1]$ is the discounting factor and he κ-value is a strictly positive scaling value.

I use the notation R_{t+1} as to show that the reward is stochastic or unknown. This is different from the r_{t+1} value which denotes an actual sampled reward at this state.

The optimality equation for Advantage learning is akin to the one for Q-learning and they are indeed equal if $\kappa = 1$. The difference therefore lies in the use of the κ-value. If this is different from 1 the Advantage value function *does not* tell us the expected return from a given state/action pair as, for instance, the action value function, $Q(s, a)$, does.

Instead, it gives a certain scaling of this value. Even though the values might be different, the relationships between the values is the same. The largest Advantage value, for instance, occurs with the same action as the largest Q-value. As we usually seek the action corresponding with the largest Q-value (or to avoid the smallest etc.) it is actually of little concern that the expected return might not be correct.

With the new scaling the differences between Advantage values may become much larger than with the action value function. If these values are somewhat uncertain (noise, function approximation etc.) this helps differentiate between them.

The optimal advantage function is of course the goal of learning, as is the case with action value and state value functions. During learning we consider the update equation

$$\begin{aligned} A(s_t, a_t) &= V(s_t) + \frac{r_{t+1} + \gamma V(s_{t+1}) - V(s_t)}{\kappa} \\ &= \max_a A(s_t, a_t) + \frac{r_{t+1} + \gamma \max_a A(s, a) - \max_a A(s_t, a_t)}{\kappa} \end{aligned}$$

Where we use the actual sampled rewards, r_{t+1}, instead of the true (hidden) reward expectation.

Notice that if the $A(s, a)$ is optimal for all $(s, a) \in S \times A$, then the numerator will be zero. This is because the future (scaled) expected return from state/action (s_t, a_t) is *exactly* the same as the actual reward plus the (scaled) expected return a time step later. Put in another way, the prediction of the future is perfect.

If the current value function is not yet optimal, the numerator will be negative reflecting that the action, a, in state s is suboptimal and leads to a less useful state. Also notice that for all $\kappa < 1$ the differences between the estimated value and the sampled value will become larger. For this reason I assume $\kappa \in]0, 1]$ is the only reasonable choice even though none of the references states this explicitly.

This reflects the main goal of accentuating the differences between the optimal value and the rest. By subtracting the right-hand side from the left we get the TD-error (Temporal-Difference error, also referred to as E^{TD}).

$$E^{TD} = \max_a A(s_t, a) + \frac{r_{t+1} + \gamma \max_a A(s_{t+1}, a) - \max_a A(s_t, a_t)}{\kappa} - A(s_t, a_t)$$

which we seek to minimize. It is assumed that $A(s, a)$ is approximated by some function approximator. The tabular situation (ie. where all function values are in a large table of size $|S| \times |A|$) can be considered a special case.

Henceforth, I consider $A(s, a)$ to be the function approximation of the real Advantage value function. This small abuse of notation embraces the close relationship between the two and is also prevalent in the literature.

The function approximator will be based on a collection of parameters, w_{ij} and these are updated at each time step, t, by

$$w_{ij}(t) \leftarrow w_{ij}(t) + \Delta w_{ij}(t)$$

where

$$\Delta w_{ij}(t) = \alpha E^{TD}(t) \frac{\partial A(s_t, a_t)}{\partial w_{ij}(t)}$$

Here I have adopted the notation, w_{ij}, for the weights in a neural network which is the function approximator I am using. α is the learning rate and $E^{TD}(t)$ is the temporal difference error at time t. Assuming that the function approximator is differential in its parameters we use gradient descent to find a local minimum. Usually, local minima are sufficient for this kind of problems.

4.1.3 Advantage(λ), adding Eligability Traces

When studying the update rule for advantage learning (4.3) (or similarly Q-learning (4.2) or SARSA (4.1)) one notices that the only reward which is involved in updating the current value is the one in the immediate future, namely r_{t+1}. This need not be so.

Consider first a task where many actions are made before rewards are received. With the T-maze I use an example would be the agent in the hallway moving north or south. Once a reward is received it will (on the next iteration) only be used to update the Q-value for the state just prior to the rewarding state. Over the cause of many episodes this information will slowly propagate back to earlier states, most importantly the starting state. This is one of the convergence certainties with RL.

One could wish, though, that the update of a state was based on rewards “further into the future”, so that earlier states “more quickly” could realize where the bounty is hidden. If the agent in my problem could see the possible bonus near the junction as early as possible, it would sooner choose to move north from the starting position. Intuitively, this ought to speed up convergence.

In general, when rewards may be given often, the agent still might benefit if it had some idea as in which direction good and bad experiences lie.

This is the motivation for introducing eligability traces which attempt to update the value function based on rewards further into the future than just one time step. Consider

$$R_t^{(1)} = r_{t+1} + \gamma \max_a A_t(s_{t+1}, a) \tag{4.3}$$

and

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \max_a A_t(s_{t+2}, a) \tag{4.4}$$

which are called the **1-step return** and **2-step return** respectively. 1-step return is just another name for the future reward we use currently. In a similar way we can define the **n-step return**.

If we used an on-policy updating technique we would consider

$$\begin{aligned} R_t^\lambda &= (1-\lambda)R_t^{(1)} + \lambda(1-\lambda)R_t^{(2)} + \lambda^2(1-\lambda)R_t^{(3)} + ... & (4.5)\\ &= (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} & (4.6) \end{aligned}$$

with $\lambda \in [0;1]$.

As can be seen R_t^λ denotes a very specific mixture of n-step returns. Many others could be conceived but this is the most commonly used. The most "immediate" return is $R_t^{(1)}$ which only require sampling one step into the future and is weighted more than any other n-step return (it is not multiplied with λ^n as the others). As we progress further and further into the future the rewards we receive accordingly become further away from our immediate decision state. These are therefore weighted exponentially smaller as they should not have too big an impact on current decision making.

Instead of the $r_{t+1} + \gamma \max_a A_t(s_{t+1}, a)$ part in (4.3) we use R_t^λ instead. Both reflect samplings of the environment but the latter gives a more refined value with regard to future rewards and their impact on the current state.

A practical use of this is of course not possible as we use rewards from a future we have not yet seen. We therefore do the opposite and tries to keep a "trace" backwards in time which plays a role similarly to the unknown future rewards. This is called the **eligability trace** and is a single variable for each parameter. The eligability trace holds the expectation of future rewards.

The approach of keeping a eligability trace of the expectation is called the **backward view** of eligability traces. Similarly, the actual R_t^λ-return is called the **forward view**. The fact that the forward and backward are actually similar is discussed in [Sutton & Barto,98] for Q-learning and SARSA. [Bakker, 02] proves the similar claim for Advantage learning with eligability traces, Advantage(λ) learning, which is explained in the next section.

A more general description of eligability traces is also found in [Sutton & Barto,98], but in this thesis it will suffice to consider the eligability trace a variable containing the update related to R_t^λ.

Since we are using an off-policy method we will not use (4.6). Instead we wish to use the optimal policy for updates and have to change it to

$$R_t^\lambda = (1-\Lambda_{t+1})R_t^{(1)} + \Lambda_{t+1}(1-\Lambda_{t+2})R_t^{(2)} + \Lambda_{t+1}\Lambda_{t+2}(1-\Lambda_{t+3})R_t^{(3)} + ... \tag{4.7}$$

with

$$\Lambda_t = \begin{cases} \lambda & a_t = \text{argmax}_a A(s_t, a) \\ 0 & \text{otherwise} \end{cases} \tag{4.8}$$

As we seek the optimal advantage value function, we should not include consequences of suboptimal exploring actions. If such an action is taken at time t_0 we zero all n-step returns from time t_0 and forward. Notice that if the action chosen one step into the future is not the optimal one, the entire trace is reduced to the 1-step return. The specific parameter update becomes

$$\delta w_{ij}(t) = \alpha E^{TD}(t) e_{ij}(t) \tag{4.9}$$

with

$$e_{ij}(0) = 0 \tag{4.10}$$

$$e_{ij}(t) = \gamma \Lambda_t e_{ij}(t-1) + \frac{\partial A(s_t, a_t)}{\partial w_{ij}(t)} \tag{4.11}$$

$$\Lambda_t = \begin{cases} \lambda & a_t = arg\max_a A(s_t, a) \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

This update is introduced by Bakker in [Bakker, 02] and is termed **Advantage(λ) learning**. The eligability trace for each parameter is updated over time reflecting the path/rewards backwards in time.

Advantage(λ) learning is the primary learning method I use in this thesis for solving the T-maze problem.

4.2 On Models and Model-free RL

When an agent acts, the consequences are unknown to the agent beforehand. We may have a good idea of a possible outcome (the next state and the reward) but generally it is uncertain.

Performing actions and learning in a trial-and-error fashion, though, might be wasteful, time consuming and/or dangerous. We may wish to *reason* about the consequences of our actions before we make them.

4.2.1 Models of a Environment

A **model** of the environment is a construct which tells us how the environment will react to certain actions without the agent actually having to make them. In chess, for example, we know where a pawn will arrive if we move it forward. A sailing boat, however, will not know exactly where it will arrive when setting the sails.

A satellite firing a stabilizing rocket is a special, real-life example. It will not know exactly where it will be at a certain time later. If, however, the engineers consider trajectories, gravitational fields, mass, etc., it is possible to predict the position with much certainty even many hours or days ahead in time. In this case all this background knowledge and calculations constitute a very accurate model of the satellites' environment.

A model incorporates all important aspects about the dynamics of the world relevant to the agent. If the agent has a model of the environment at its disposal, it can reason about effects of actions by consulting the model to predict consequences. Theoretically, the model is the *distribution* on which the world is based. The actual environment, or sensation, is a specific *occurrence* drawn from this distribution. Models are formally defined as the "transition probability" and the "expected reward".

The **state transition probability**,

$$\mathbb{P}^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

denotes the probability of arriving at state s' when the agent takes action a in state s. In the T-maze this is a deterministic function for the most part. But when escaping the maze it is random what starting state (ie. road sign) the agent will arrive at.

The **expected reward** is defined as

$$\mathbb{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

Even though rewards are often fixed, they may also be of stochastic nature. Sutton [Sutton & Barto,98] describes an example of an agent working a slot-machine. The reward here will be stochastic no matter what the agent learns. In general, rewards can have any distribution but the distribution itself is actually not necessary for the agent. The mean expectation is sufficient.

During training the rewards might appear random to the agent which only has a local view. In our problem, for instance, the act of moving east in the junction will result in either -0.1 or 4.0 but this cannot be deduced from the junction state and move action itself. As the agent learns to incorporate the road sign into the junction state, the reward becomes deterministic to the agent as well.

Learning with a model

In model-based learning the agent can use the state value function, $V(s)$, by examining the expected value of each state it could visit. This makes sense as the model can explain where each action will take the agent. With no model at hand, however, the state value function looses its use as the agent has no way of knowing what state a specific action will lead it to.

As an example, consider the game of backgammon. The player knows beforehand the probability distribution of the two dice and can therefore reason about any of the future possibilities in the game without actually having "been there". The rewards in themselves may be very simple. In backgammon we might give a reward of 1 for winning the game, -1 for losing and zero for all intermediate state transitions in the game. In theory, we might from any state reason out all the different ways of winning along with their probabilities which would sum to the state value, $V(s)$, of the state.

Note that both chess and backgammon (and even checkers) share a combinatorial explosion in computation. So knowing the exact effect of actions does not in itself make life much easier. Still, it makes it possible to reason about future consequences and this is employed by most software-efforts to play these games.

In my problem a model is easily made. If for example the agent is in the hall and moves north, it will with small probability reach a junction and with large probability a hall-state again. When in the junction, a movement to the east will lead to the beginning state for certain (varied by the road sign) but the expected reward will be 3.9 (mean of 4.0 and -0.1).

4.2.2 Learning with no model

Most games are problems with a model. It is perhaps part of human nature to prefer leisure-time activities where the only "hidden" and random factor is the human opponent. In contrast, most real-world tasks are problems with no discernable model. Two approaches exist in model-less problems, either to *learn a model* or to use *model-free* learning.

When **learning a model** the agent essentially moves about experiencing the environment and tries to build its own approximations of the state transition probability distribution and the expected reward. In theory it has to visit all states, take all actions and notice all rewards an infinite number of times.

In practice, a substantial uncertainty is usually acceptable and the agent can begin using model-learning in the same way as mentioned above. The learning process is therefore essentially two-phased. First the agent learns a model and then moves on to model-based learning.

In **model-free** learning the agent never bothers to build an explicit model of the environment. Instead of the state value function, $V(s)$, the agent uses the action value function, $Q(s, a)$, and therefore samples the consequences of a specific action from a specific state. While this eliminates the task of explicitly building an often complex model it introduces other problems which usually must be dealt with.

Seemingly, everyone should prefer the action value function instead of the state value function as the former can solve all task the latter can. Still, even with a modest number of actions the action value function-table will be many times the size of the state value-table. Even more importantly, the learning is prolonged as we do not sample states but state/action-pairs which will require more time and data.

4.3 Exploration Issues

Primarily the notion of **exploration** and **exploitation** plays an important role in model-free learning. Suppose the agent stumbles on a series of state-action pairs which leads it to the goal. There is no reason to believe this path will be an optimal one. By *exploiting* this path the agent is certain to receive some reward (provided the environment does not change). By *exploring* other actions it may find a better way. If little exploration has been done it is likely that a better path can be found, if the agent has explored a lot it is probably better just to stick with the best solution found so far.

This is a more important matter than it might seem. Even in problems where any solution is acceptable (like any means of winning the T-maze) pure exploitation may be inefficient and may even prove hazardous.

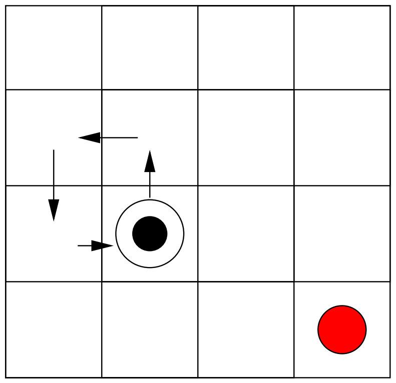

Figure 4.1: An agent (large circle with a smaller filled circle inside) moves about a grid world. Moving into a wall gives negative reward and the goal (large filled circle) gives positive reward. Everything else gives zero reward. From random sampling the agent expects all actions to lead to negative rewards and therefore gets stuck in a sub-optimal zero-value path.

In figure 4.1 the agent receives zero reward moving about, a negative reward for hitting walls and a large positive reward for reaching the goal. As it initially have no idea where the goal is it moves about randomly and hits some walls in the process. The result can easily be this: the agent assumes that most actions lead to negative results but finds a loop which has a zero reward. It gets stuck in this loop forever as these precise actions in the four states have the optimal expected reward.

This situation might seem artificial and stems from some obvious flaws in the design of the agent, especially the reward function (The reward function is technically not a part of the problem, it is part of the solution or the solution technique.). It is not clear, however, if one has designed the agent and the reward function sufficiently correct, so that such situations can be avoided. It is not clear either, whether it really is a design problem or not.

During the working with this thesis and especially in earlier work with continuous domains [Nielsen & Hansen, 05] I have noticed similar strange behavior. Sometimes the agent ends up just spinning around itself (turning was a zero-reward action), sometimes it moves back and forward through the same path. I have even seen it converging to a behavior where it repeatedly moved directly into a wall! This radical behavior also proved to stem from problematic exploration along with a flawed reward function.

Even if it hits the goal by chance it may over time "drown" this reward in a sea of negative feedback, especially if the environment is so complicated that the agent perceives it as random or chaotic experiences.

RL theory promises us that given some rudimentary prerequisites, the agent will eventually learn the solution, even the optimal one. This, however, is only an asymptotic certainty and it may require prohibitingly much training data and/or training time. Measures to speed up convergence and help learning are essential. Also, if the state- and action space is not small and finite, we will be forced to approximate value functions and thereby lose the promises of convergence.

Some exploration is generally needed, not only to converge to optimal behavior but also to facilitate learning. This will prove necessary in our T-maze problem as well, even though it seems trivial compared to real-world problems. With the need for exploration established, the next questions are "when" and "where" to explore.

4.3.1 Directed or Undirected Exploration

The most simple exploring policy is the so-called **ϵ-greedy** approach. The most promising action is usually chosen (greedily) but with some probability, ϵ, a random, suboptimal action is chosen instead. This way, the agent cannot get stuck in a suboptimal loop forever as it with a some probability will search other parts of the state space. The ϵ parameter can change over time; maybe decaying towards zero so a good, stable performance can be achieved.

This exploration policy is **undirected** as it does not favor any part of the state-action space to explore. Another undirected technique is the **Boltzmann distribution** (also Gibbs-Boltzmann in some references). For the state, s_t, the next action is chosen from the probability distribution

$$P(a|s_t) = \frac{e^{Q(a,s_t)/\tau}}{\sum_A e^{Q(a,s_t)/\tau}}$$

where $\tau > 0$ is the **temperature**, A denotes the action space and $Q(s,a)$ is the action value function. The denominator sums over all actions possible in state s which normalizes the numerator to a probability.

If the temperature is large all actions will become equally likely to be selected. If the temperature is small it will approach a greedy policy since the maximum Q-value will generate highest probability. [Sutton & Barto,98] states that it is unclear and probably problem dependent which of the two exploration techniques are better.

Neither method, though, has any clear preference when it comes to the exploring action, even though the Boltzmann distribution will favor exploring actions whose Q-value are higher.

A **directed** exploration technique choose exploring actions which are believed to be worth taking. It therefore seeks to focus exploration on the "right" parts of the unknown state space and stay away from the "wrong" parts. This will of course always be preferable to undirected exploration but it is more difficult to design and conceive than regular exploration. So the difficulties must be worth the effort.

Consider as an example a suboptimal action which has is large Q-value which is fairly certain. This would happen when it has been sampled many times which reduces the variance of the estimate. In this case it might be a better choice to select an action with even lower Q-value but with more variance in the expectation. This reflects that the latter action has not been explored much whereas the former probably has been explored more with unsatisfactory results (as it has not become the optimal choice).

Below I present another choice: Bakker's directed exploration method.

4.3.2 Directed Exploration based on TD-error

Bakker's directed exploration-technique is presented in his dissertation [Bakker, 04]. It is based on the Boltzmann distribution presented above but integrates an estimation of the TD-error computed at each time step. I remind that the **Temporal Difference-error (TD-Error)**, E^{TD}, denotes the difference between the current estimation of the future return and the actual sampled estimate. For Advantage learning this would be

$$E^{TD} = r_{t+1} + \lambda \max_a A(s_{t+1}, a) - A(s_t, a_t) \tag{4.13}$$

The TD-error is a measure of how well the action value function estimates the future rewards. If this error is small the agent can be fairly certain that its estimate of the world is correct not matter how large or small the Q-values are. Therefore, the agent should be more inclined to make a greedy choice in this situation as the action value function can be considered reliable. If the TD-error is large, however, it reflect an inconsistency in the action value function; that it cannot reliably estimate the consequences of the actions.

As the value function is all we have at our disposal it is necessary to use it anyway to select an action but the high TD-error reflects that it probably would be beneficial to explore more in these situations.

To do this a small feed-forward neural network is created which estimates the absolute value of the TD-error at each state. Its input is the same as the input to the action value function which in my problem is three binary neurons. One hidden layer with 6 sigmoidal neurons seems to work reasonably well. One output neuron with identity activation will at each state hold the expectation of the TD-error.

This error estimate then plays the role of temperature in the Boltzmann distribution. The directed exploration probability is

$$P(a|s_t) = \frac{e^{\frac{A(s,a)}{C \cdot E^{TD}}}}{\sum_A e^{\frac{A(s,a)}{C \cdot E^{TD}}}}$$

Where C is a scaling constant. This distribution tries to complement the benefits of a Boltzmann distribution and a focus on the TD-error size. Notice the use of the Advantage Value function, $A(s,t)$, instead of the action value function, $Q(s,a)$.

If all advantage values are similar the probability will be a uniform distribution. If one or more advantage values are greater than others they are more likely to be selected. Both traits are desirable as we wish to explore more if no action seem promising and explore less if some actions seem to have a low advantage.

If the TD-error is low, the temperature will drop and the largest advantage, $\max_a A(s,a)$, will dominate the smaller ones. This will make the distribution approach a greedy one as the large advantage will acquire all the probability mass.

If the TD-error is large, the individual probabilities will flatten and make the larger actions less probable of being selected in favor of the smaller ones. As the TD-error grows, the distribution will approach a uniform distribution. This will make the agent seek to find a way out of the high TD-error situation.

Some exploration will always occur even for very low TD-errors. This may be desirable to escape suboptimal extrema or if the environment changes over time. Otherwise, a thresholding device can be employed to fixate the policy once the exploration is too low.

Bakker actually does the opposite and ensures that a minimum of 5% of all actions selected are exploring actions (ie. not choosing the action with optimal value). This is to prevent the exploration scheme from decaying to an almost-greedy policy too fast. Even with the TD-error temperature, the action value function can still approach something like a greedy decision making the right path harder to spot.

4.4 Summary and Reflections

In POMDP problems, especially real-life ones, it is worth considering using model-free learning. As a model of the world is seldom available, building one's own model is time consuming and might be especially difficult if there are underlying temporal relationships in the world.

Advantage learning sidesteps this by letting the agent learn and interact directly with the environment without building a model explicitly. Compared to the common off-policy method Q-learning, Advantage learning is designed to handle noisy situations, especially function approximation, better.

Advantage(λ) learning is introduced, adding eligability traces to Advantage learning. Eligability Traces usually bring a benefit in reduced convergence time to an RL-system. As I will return to in the experimental part, it does not work quite as expected. Advantage learning, though, is clearly preferable to Q-learning.

Directed exploration may play an important part in POMDP and I have introduced Bakker's directed exploration technique. Below I will comment on this further.

4.4.1 Comments on Directed Learning and POMDP

I have mentioned that directed learning is what "we have always wanted" but the troubles of developing/using it might be too much trouble. In POMDP-settings with function approximation, though, I believe they generally are a necessity.

As Bakker notes, POMDP problems usually have a large *majority* of unambiguous states. Once these have been clarified there is less reason to explore. In my problem, the T-maze, this could be the hallway: movements to the east and west will lead to punishment, movement to the south does not really lead to anything useful, movement to the north may lead to a high reward. Once this is learned the TD-error will be low and less exploration is needed.

This is contrary to the junction, or generally any state which is ambiguous. Here it must be assumed that the non-markovian aspect of the state will make the agent choose more wildly and is bound to make inappropriate choices. Therefore, the estimates will be off and the TD-error higher. As ambiguous states are the most crucial ones to explore in a POMDP, Bakker incorporates the TD-error in his exploration. Once the ambiguity of the state becomes solved through the use of memory the TD-error will drop as well.

The ambiguous states may need thorough exploration before the temporal dependency is realized. If this exploration is not done, the agent might get stuck in some behaviour which will *keep it from seeing the temporal relation*, as it will not see the correlation between road signs and wins.

One point is to limit exploration in states where it is not necessary and keep exploration high in states where it is important. While exploring in important state is obviously a good idea, the act of *not* making unnecessary exploring actions might be equally important. These actions will introduce unnecessary noise in the system, making the fine temporal correlation harder to spot. It is important that the agent learns to move very directly to the junction. To much walking around in the corridor and into walls keeps it from learning the solution. This problem can in short be described as *when* to explore

Another point is *where* to explore whenever it is done. This is probably less important in the unambiguous states, but in the ambiguous states there will be a few promising actions and the rest would be less useful. In the T-maze problem the agent should keep its search focused on the east and west options as they will have the two largest Advantage values (their mean reward is 4.0+ -0.1 = 3.9 much larger than south (0) or north (-0.1)). In general the agent would over time learn that these selected states have *potential* and that the others are less useful. As the temporal relation is hard to learn it is important to sample the east and west action more than north and south.

4.4.2 Bakker's Directed Learning Technique

Both of these features are present in Bakker's directed exploration technique and for this reason I consider it extremely suitable for problems like this. A drawback may be that the technique is a somewhat "slow learner".

Usually "realization" in RL comes in bursts: once the agent discovers that walls entail punishment, it will "suddenly" avoid walls altogether. Also, when it finds a clear path to the goal, it will dramatically improve performance in a very short time. In one particular slow run on the T-maze the agent ran through 335,000 episodes not realizing whether to turn left or right at the junction; thus performing at chance level. Over the course of 500 episodes, once the correlation between goal and road sign became apparent, the performance rose to perfect.

Bakker's directed exploration, though, will need many visits to the state after "realizing" what the correct action is before the TD-error will have dropped significantly. This will lead to many "superfluous" exploring actions in this state. A similar effect will take place the first time a state is visited in which the network will assign a somewhat random TD-error to this state, making exploration a random issue as well. This issue stems from the use of function approximation of the TD-error, in this case a neural network. New states will simply have a TD-error close to the known state it is most similar to.

If such slow convergence of the TD-error estimator is acceptable the technique can be very useful. As RL-learning in itself is converging slowly over time as well (except for the "bursts"), I believe the TD-error estimator would fit most applications.

Advantage(λ) Learning and POMDP

Usually, the introduction of eligability traces is "only" to reduce convergence time. In elementary RL they neither help nor hinder the fact that a solution is learned or not.

In POMDP, though, this might be different. As the network will converge to some extrema it is important that this be a useful one. If too many episodes are needed in order to let rewards propagate back in the network, the network may itself get stuck somewhere where it cannot get out. Therefore, it is likely that the eligability trace may help in solving the problem.

The opposite may also be possible. Perhaps the introduction of these traces makes the reward-signal too “blurred” to understand. I return to this in the experimental section.

Chapter 5

Recurrent Neural Networks, prelude to LSTM

To set the stage for LSTM I will in this chapter introduce the Recurrent Neural Network (**RNN**) and its relationship with the Feed-forward Neural Network (**FNN**). The size of RNN literature is impressive though not as immense as the coverage of FNN. Still, I only cover those essentials which are relevant to LSTM.

RNN entails some tantalizing benefits but also some devastating drawbacks. In order to fully understand these, I first describe RNN along with its training. With an understanding of the inner workings I turn to the pros and cons of the technique.

In the next chapter I move on to LSTM which is an RNN structure that addresses these problems. The formulation was augmented by Felix Gers in his dissertation [Gers,01] with the so-called "forget gates" which are also discussed here.

5.1 Basic definitions

An RNN is a regular neural network where at least one neuron has a connection "backward" in the network so that, at some point, it will receive its own output as input. RNN is not a new development, it was considered a natural extension to FNN in [McLelland, 86] which is close to the time where FNN itself became widely known.

RNNs are also known as **feed-back neural networks**, **folding neural networks** and probably other names as well. Sometimes the different names denote specific designs or restrictions but I do not imply any such restrictions and will just speak of RNN in a more general setting.

5.2 Learning with RNN

When training (or setting the edge weights) FNNs the two dominant approaches are genetic algorithms and backpropagation. Below I describe these when applied to RNN training and point to their benefits and drawbacks for this task.

5.2.1 Genetic Algorithms and RNN

In a **genetic algorithm (GA)** one operates with many different networks for the same problem. Each network is called an **individual** in the larger **population** of networks. For each individual network all edge are fixed before execution and are not changed during exposure to data.

During testing on data some error is calculated, usually referred to as the **fitness** of the system. With this fitness value the network is compared to other networks. The better ones are kept and poorer ones are eliminated. This constitutes one **generation** of individuals. A new generation is created based on the old one through a complex scheme involving mutation, crossbreeding and similar processes inspired from biology.

From a GA point of view there is little conceptual differences between RNN and FNN when assigning edge weights. The recurrent edges are just an additional set of edges.

Still, the recurrent connections play a more fragile role than the other connections in the network. In a regular FNN there is some “leniency” in the setting of the edge weights. For instance, in some FNN all edge weights can be multiplied by the same constant without changing the results. This particular feature stems from the fact that a separating hyperplane is defined only down to a scalar multiple.

But multiplying or even slightly changing a recurrent connection can cause more dramatic results over time. Thus, in the population-space that GA works with it might be more difficult to decide if two RNN are similar, how to crossbreed them reasonably, how to mutate, etc.

GA can, of course, be used with RNN-learning with decent results as [Ziemke, 99] shows. Also, GA evades the learning problems which cripple backpropagation. These problems are explained in section 5.4 but generally stem from difficulties with gradient descent through the recurrent loop. Such gradient descent is not performed in GA which just discards ill-behaved networks.

I do not use GA in this thesis. It does not have the more clear path through weight-space toward a decent minimum as gradient descent has. Even though GA usually converges to some local minimum, it does this in a more random and uncontrolled way than backpropagation. This lack of control makes it difficult to explain unsuccessful training and this might be critical in the more complicated LSTM network.

The interactive nature of RL and the T-maze also makes it more suitable for backpropagation. Finally, I wish to build on LSTM and Bakker’s results which entail this learning method.

5.2.2 Backpropagation for RNN

Backpropagation is a popular approach to learning in FNN. By adjusting edge weights with regard to the produced error (in a single sample or a batch) backpropagation performs a gradient descent in weight space towards a local minimum which often seems to work reasonably well.

It does, however, not incorporate recurrent networks. In essence, the recurrent connection "screws up" the error-propagation algorithm as it tries to induce error into a neuron which has already received error and adjusted its weights. The two main approaches[1] to incorporating recurrent connections are called **Backpropagation Through Time** (**BPTT**) and **Real Time Recurrent Learning** (**RTRL**). Both are important to LSTM as they play an integral role in LSTM.

Backpropagation for RNN with "Backpropagation Through Time"

The essential idea in BPTT is to "unfold" the folded structure of the RNN. [McLelland, 86] notes that for all RNN there exists a functionally equivalent FNN if viewed over a *finite* period of time.

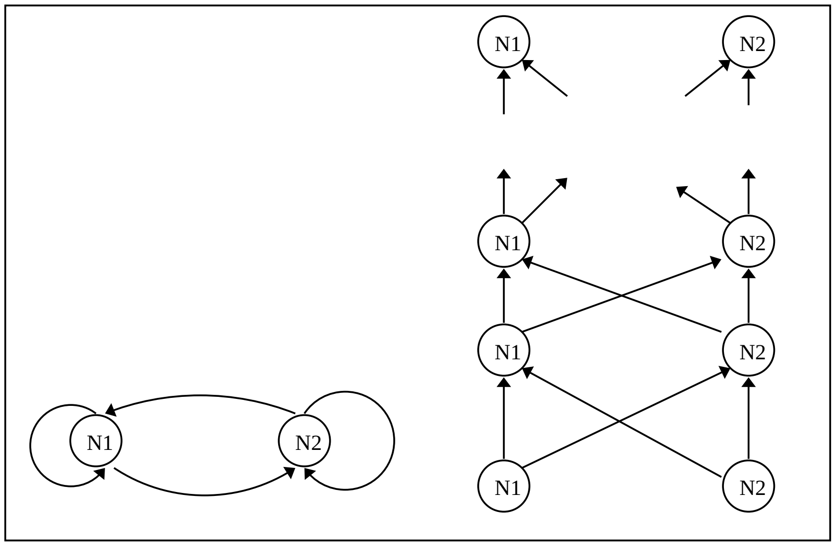

Figure 5.1: Backpropagation Through Time (BPTT) as it is seen in [McLelland, 86]. The recurrent network on the right is "unfolded" to a similar feed-forward network with a depth equal to the time step, t. Backpropagation is carried on conventionally and the sum of weight changes is added to the original weights. Notice that the same two neurons, N1 and N2, are "copied" t times.

Consider figure 5.1. On the left is a recurrent network on the right the "unfolded" corresponding feed-forward network at time t. Notice that for

[1][Williams & Zipser] present numerous references on variations/special cases of these two.

each time step we introduce a new neuron for each recurrent connection. These neurons are then connected with edges which are equivalent to the original feed-back edge.

In order to backpropagate an RNN at time step t, we simply unfold the recurrent network in a t-layer FNN and perform standard backpropagation. The final weight change to the original edge, w_{ij} will be the sum of all the unfolded w_{ij} "weight changes".

This is obviously a challenging task for long episodes and will break down in continuous tasks. Mainly, the memory requirements are considered the drawback, as they are substantial.

Truncated BPTT is an important variant of BPTT with a rather obvious property: it simply stops the backpropagation at a fixed depth of the unfolded FNN. The choice of depth is not constrained and we will in LSTM see a specific use of this. This makes BPTT more accessible to problems with longer time lags but as soon as the important information grows beyond the depth of the network it will no longer play a role in updates and the network may break down.

Backpropagation for RNN with "Real Time Recurrent Learning"

BPTT has the obvious problem of the growing unfolded network over time. In the literature this is referred to as a memory problem even though it might entail computational problems as well.

RTRL was conceived to counter this growing property. It is devised in its most general form by [Williams & Zipser] (for a more thorough description see [Williams & Zipser]).

Define

$$p_{ij}^k = \frac{\partial y^k(t)}{\partial w_{ij}} \tag{5.1}$$

Here, k denotes the output of any neuron at time t. Let $T(t)$ be the **target vector** describing the target, d_k for neuron k at time t. It is time varying which allows for different neurons to have different targets, or targets at all, over time. Consider an error measure of the form

$$e_k(t) = \begin{cases} d_k(t) - y^k(t) & \text{if} k \in T(t) \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

Here any neuron might have an error which is used for updates. This is more general than in our setting where only output neurons have targets. Next, we consider the error function at time t

$$J(t) = \frac{-1}{2} \sum_{k \in U} [e_k(t)]^2 \tag{5.3}$$

Which is the function we seek to minimize. To do this by gradient descent we find

$$\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) p_{ij}^k(t) \tag{5.4}$$

When applying this to the activation a time step later we derive

$$p_{ij}^k(t+1) = f_k'(net_k(t+1))\Big(\sum_{l\in U} w_{kl}p_{ij}^l(t) + \delta_{ik}y^j(t)\Big) \qquad (5.5)$$

$$p_{ij}^k(0) = 0 \qquad (5.6)$$

Where the δ_{ik} is the Kronecker delta function.

Notice that the update of the partial, p_{ij}^k, is dependent only on the value of the previous time step. This is different from BPTT where the weight change is computed by unfolding all the way back to the first time step. RTRL is therefore not as precise as BPTT but gives a good approximation of the partial computed by BPTT.

The main drawback of RTRL is a computational one. In general, the p_{ij}^k need to be stored for all values of k, i and j which is conceptually done in a $N \times N \times N$-matrix where N is the total number of neurons. This is necessary as it is unknown when or if a specific neuron will receive an error signal, $e_k(t)$.

For fully connected networks [Williams & Zipser] notes that the approach has a space complexity of $O(n^3)$ and a computational complexity of $O(n^4)$. If one's network is not fully connected or just a restricted subset of neurons will ever receive error, this can be reduced.

The problem of changing weights at each iteration without taking steps further back in time is that we deviate somewhat from the true error-gradient which needs all steps back in time. The idea is then to make small changes to the system so that the difference between RTRL and the true gradient is thought to be "small". This idea is similar to making incremental updates in an FNN instead of updates based on a batch of samples.

5.3 Benefits of Recurrent Networks

The introduction of the feed-back connections gives the RNN a more complex structure which again gives rise to added computational power. It is obvious that FNNs are a subset of RNNs, so all problems solvable by an FNN is solvable by an RNN. This might motivate the use of RNN and there is at least three points which make RNN preferable to FNN.

1. RNN can solve some problems which FNN cannot.
2. RNN works better on some problems than FNN.
3. RNN mirrors some biological structures (neuron/synaptic connections in brains) which FNN cannot.

The first reason is the main use in this thesis but they may all have merit depending on the situation. I elaborate on these below but begins with the second reason as it naturally leads to the first.

In the following I assume that no “learning” is taking place. That is, no weights are changed at the different time steps. This way, the only changing variable over time is the input vector. The main ideas presented below are still valid in a learning situation as the edges only change “slowly” over time compared to activation values. The argumentation, though, becomes more involved.

5.3.1 Problems where RNN Will Work Better than FNN

No matter how complex an FNN, its output will always be completely determined by the input vector at the current time, t. Even if we momentarily include learning (changing of edge weights) we still cannot fully capture the information from previous time steps. The edge weights change too slowly and too imprecisely.

As FNN, RNN output is also determined by input vector but also by all previous activations at time t-1, t-2, etc. Thus, RNN will have an advantage if there are temporal regularities which can be of use. A classic example is the ’N-bit parity checker’ see figure 5.2. Given an 8-bit input string, we wish to see if an even or odd number of these bits are set.

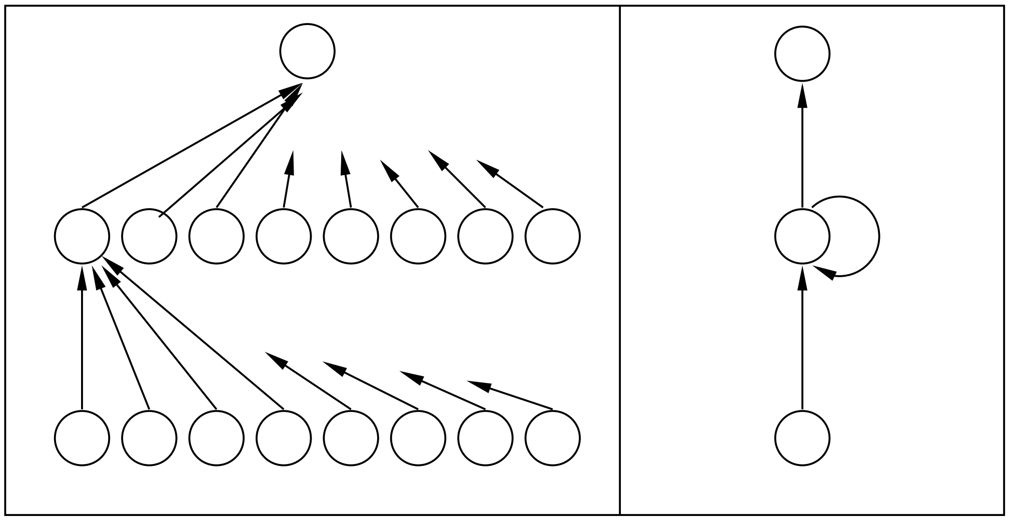

Figure 5.2: The 8-bit parity problem. If the number of set bits are odd the output node has value 1 and otherwise zero. With an FNN this needs a large and complicated structure but RNN can solve it much more easily by exploiting the temporal relation.

In a regular FNN, first of all we need an 8-neuron input layer. Also, since the problem is not linearly separable, we need at least one hidden layer. The FNN shown in figure 5.2 is from [McLelland, 86] totalling 17 neurons and 72 connections. This does not scale well with increasing N.

In an RNN just a single recurrent connection is needed. Basically, the recurrent connection switches on and off according to the current parity. If an even number of bits has been processed the output will be zero for as long as no active bits are shown. If another active bit is introduced the recurrent connection will get the status 1, and the output will be 1. This network will not grow as N grows.

In essence, the parity bit problem is a temporal problem. We wish to *count* a specific property. As the FNN cannot resolve such temporal problems it needs the entire history of input in order to compute the output.

This simple problem shows an important point: A problem with temporal regularities might be easier to solve if these regularities can be used efficiently.

Bakker [Bakker, 04] expands on this problem by training an agent to go either left or right at the end of a T-maze somewhat similar to mine. At the junction it receives an n-bit parity problem (Bakker made several tests with varying degrees of difficulty) which solution is the right way for the agent to turn. Also, in the corridor just prior to the junction, the agent receives a less complicated m-bit parity problem which solution is also the correct turn to make at the junction.

The agent had both a recurrent network and a regular FNN at its disposal (kind of "in parallel") but it had to choose which of the networks it would use to make the decision. Bakker showed that when the junction-problem was difficult and the corridor problem was easy, the agent chose to use the recurrent network. In this situation the agent decides that the temporal information was easier to use. If the corridor problem was difficult but the junction problem was easy, the agent similarly chose to use the FNN to make its decision.

In this experiment Bakker showed that even though the problem could be solved with a FNN, the agent learned to use a recurrent network instead, as it learned that the temporal regularity was easier to exploit. This is a remarkable result as not only do the FNN and RNN learn to solve the problem but the *decision* of which one to use is also learned.

5.3.2 Problems which FNN cannot solve, but RNN can

Given the discussion in the last section, it is easy to see where FNN comes up short. In the parity-bit problem, the FNN can never produce a correct solution if it has a single input neuron and is fed the bits one at a time. The entire history is needed. But histories tend to make data explode.

Consequently, the network is forced to grow over time. The problem is not only that the number of input neurons will grow indefinitely, learning or setting the correct edge weights for the incoming neuron is not easy, not to mention the changes to the rest of the system. We could amend by giving the network a finite "window" back in time (where the last n samples back in time is used as input) but it is obviously a limited solution. As soon as the relevant data exits the window the FNN breaks down to random performance.

Another related problem is that an FNN will have severe problems discerning which parts of the history it needs in order to compute the correct result. A large parity problem (perhaps thousands of bits) would create huge learning difficulties if most of the bits were zero.

The proposed recurrent solution does not have this problem, but *this is only because it is relatively easy to pin-point the relevant data in history.* In general, RNN are prone to the same problem of filtering out important data from the past. Essentially, this is a problem for any learning system.

5.3.3 Biological Parallels

Whereas the cellular function of brain cells are fairly well known, the emergent properties of these are much less revealed. Not, though, through lack of effort on the matter, but the sheer size of the neural network in the organic brain prevents conventional investigations.

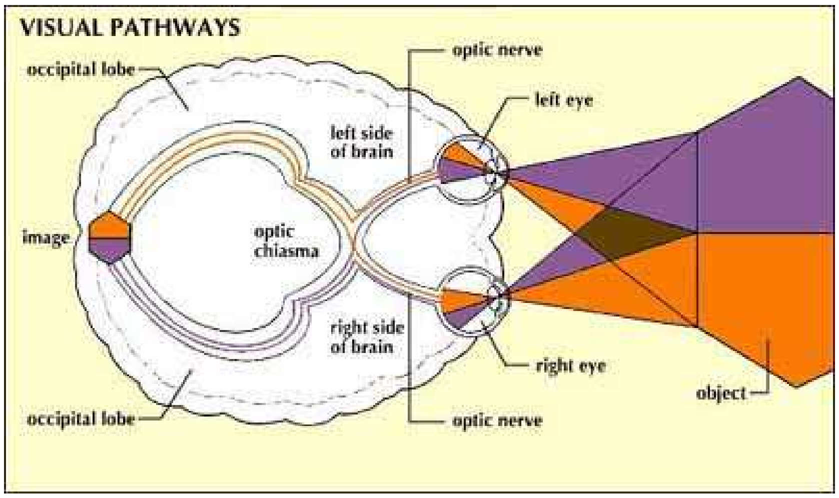

Figure 5.3: The picture shows the neural pathway from the eyes to the higher cortical structures. Notice that the path leads all the way back through the head to the most posterior part of the brain: the Occipital lobe. The first sub-structure of the Occipital lobe that the pathway connects with is termed V1. Up until this point the pathway is predominantly feed-forward. From V1 and further into the cortex the processing is almost completely a blended feed-forward/feed-back structure.

A muscle, or a bone, might have the same number of cells as some area of the brain but the inter cell relationship is much more clear in a muscle and is often of lesser importance.

In the brain, on the contrary, a single cell might in theory (and probably also in practise) change the behavior of the entire system. All the interesting processing in the brain relies on the inter cell relationship between the neurons rather than the neurons themselves.

As an example consider the neural pathway from the eyes to the visual cortex in the human brain. The beginning of this pathway is one of the few areas of the central nervous system which has a feed-forward structure. Before reaching the occipital lobe this feed-forward signature is prevalent even though some feed-back connections exist. From the V1 area and forward, though, the computation is highly integrated with multiple feed-back connections [Gazzaniga, et al. 02]. From then on, our understanding of visual cognition is on a *highly* abstracted level.

Therefore, if for no other reason, the study of RNN is necessary in order to simulate and/or understand these structures in the brain.

5.4 Problems with RNN

The greater computational power of RNN is tantalizing and the above mentioned examples show that interesting solutions can be made. Alas, RNN has severe drawbacks.

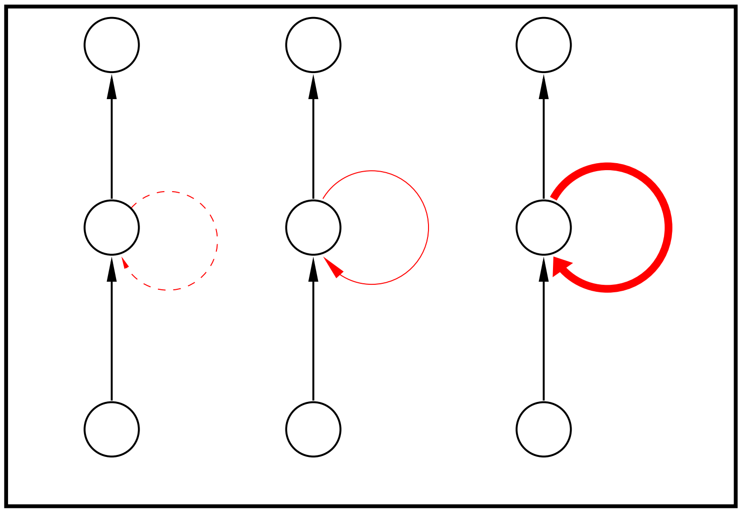

Figure 5.4: A somewhat simplistic but intuitive visualization of the RNN 'noise-buildup'-problem. Initially the recurrent connection is empty but over time it will keep recycling all input to the neuron. As it cannot "let go" of this, over time it will grow so big that it will dominate input and the connection will lose its functionality.

The recurrent connection tends to built up noise as time passes and this often make the recurrent part useless rather quickly. This problem has an intuitive explanation: In the beginning the recurrent connection is empty (ie. has zero input to the neuron). As time goes by the neuron will keep receiving new input as well as input from itself. It just keeps adding the information from each time step to itself.

Essentially it tries to remember everything it has ever "seen", indiscriminately. If human beings were to do the same thing we would probably not survive more than a few seconds before our brain was overloaded. 5.4 illustrates this with an assumed identity activation function for the neuron. The choice of activation function does not help much, though, the problem is also evident with the sigmoid function. Essentially, the problem is not so much that the cell state or the recurrent edge diverges but that the information it is supposed to keep is “corrupted” with noisy input.

Another problem is backpropagation with RNN. [Hochreiter, 91] shows that the propagated error in networks when trained with either BPTT or RTRL (see 5.2.2 and 5.2.2) will have exponential behaviour: It will either blow up or vanish fast. To reset the connection now and then seems necessary, but this again limits the RNN to uses with just short time lacks

5.5 Summary and Reflections

RNN has at least three features which makes them superior to FNN: RNN may work where FNN does not, RNN may outperform FNN on specific tasks and RNN mirrors neural structures that FNN cannot.

Still, RNN has drawbacks which are all related to its affinity for “storing” or reacting to everything it has ever seen. One aspect is that the recurrent connections tend to build up noise over time. In the extreme cases the recurrent connections may blow up and diverge but even in the general case the connection will disrupt their sensitive contents over time.

Another, perhaps more important, aspect is that training of RNN is difficult. With backpropagation [Hochreiter, 91] shows that the two most accepted kinds of backpropagation, BPTT and RTRL, will be subject to an exponential behavior of their propagated error. Either these will blow up or vanish rather fast.

Chapter 6

Long Short-Term Memory

[Schmidhuber & Hochreiter, 95] proposed a solution to the problems of RNN learning which I will describe in this section. I begin with **traditional LSTM** as introduced in [Schmidhuber & Hochreiter, 95] and then integrate **extended LSTM** which adds "forget gates" to the system. Extended LSTM is a development by Gers, [Gers,01], and also includes "peep-hole connections" which will not play a role in this thesis.

The technique shows astounding results and has been able to breach time lags as long as 1000 steps back. In most or all reported cases LSTM clearly outperformed conventional RNN techniques (and, of course, FNN-solutions) sometimes by several orders of magnitude.

Updating, or training, an LSTM network is "local in space and time", meaning that it does not have the memory explosion of BPTT nor the large computational requirement (the large network matrix) of RTRL. Thus, it scales well. LSTM does employ both techniques but in a restricted fashion.

The theory behind LSTM is not easy to come by but with an intuitive understanding of the problems mentioned in the previous chapter, I cover the subject in a piece-by-piece fashion.

Firstly, I summarize the notation used in this chapter. Next, I describe the network topology which I employ in my agent and will base the remaining discussion on that. Next, I describe the connections in the design and the activation of the network (what Gers refers to as the "Forward pass"). The third section describes the learning algorithm, or backpropagation, for LSTM (similarly "the Backward pass"). Lastly, I comment on the technique and its relations to my problem.

6.1 Indices and Notation

In this thesis the following notation will be used concerning the LSTM system. It is based on standard neural network notation and LSTM-notation with some additions to increase clarity.

The system is made of neurons connected through directed edges with a real-valued weight. The edges have a source neuron, x, and a destination neuron, y. The connection between these are traditionally indexed somewhat contra-intuitively as w_{yx} with the destination neuron first. I use the following indices:

- "o" indicates an output neuron. In this thesis the output layer consists of one neuron for each action so further indexing is not necessary. If I need to differentiate between actions, it will be clear from the context.
- "h" denotes hidden neurons in the FNN part.
- "i" denotes input neurons.
- "c" denotes cells.
- "iG" denotes ingates.
- "oG" denotes outgates.
- "fG" denotes forget gates
- "m" will be used to denote memory blocks.
- "j" and "k" will be used to denote arbitrary neurons whether it be input, hidden, output, cell, etc.

The sigmoidal functions used in this thesis are defined as

$$\begin{aligned} f(x) &= \frac{1}{1+e^{-x}} \\ g(x) &= \frac{4}{1+e^{-x}} - 2 \\ h(x) &= \frac{2}{1+e^{-x}} - 1 \end{aligned}$$

With $x \in \mathbb{R}$. f is the activation function used if nothing else is mentioned. Notice the range of these functions which are $f(x) \in]0,1[$, $g(x) \in]-2,2[$ and $h(x) \in]-1,1[$ respectively.

6.2 Network Design

LSTM is a powerful function approximator capable of handling time dependencies. In RL in a POMDP setting, a natural place such a function approximator is to approximate the value function. In our case this is the Advantage(λ) value function.

The network we wish to conceive must be able to handle two types of situations: those without time dependence and those with. The first situation is the well-known Markovian one and may fruitfully be solved by an FNN. The second one involves "memory" and will be solved by a system of memory blocks. A **memory block** is the only part of the network where recurrent connections exist. The rest of the system has no direct connections to the parts where the recurrent connections reside only indirect connections through the memory blocks.

The network will consist of an FNN-part and a memory block-part as can be seen on figure 6.1.

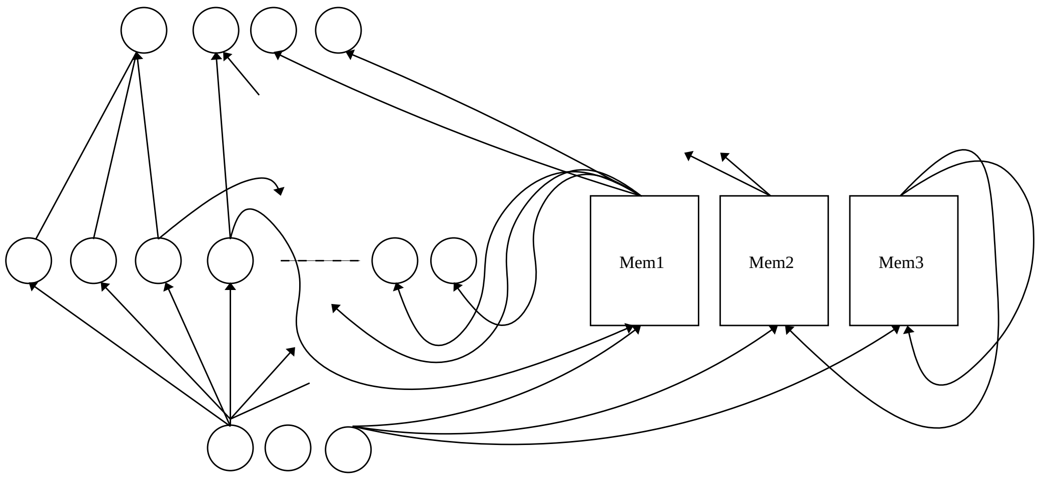

Figure 6.1: The design of the network I use. To the left is a regular FNN with just one layer. Each output neuron is attached to just three neurons in the middle layer. To the right are the memory blocks which hold all recurrent connections. These receive input from all input neurons, all hidden neurons and all memory blocks (including themselves). Hidden and output neurons receive input from all memory blocks as well.

Here, the network has a seemingly parallel buildup: FNN to the left and memory blocks to the right. To some extent this mirrors the functionality of the system. When the agent is in non-ambiguous states the FNN part is able to infer the Advantage value directly and does not need to use the memory. When the agent is in an ambiguous state it uses the memory blocks as well.

This division is somewhat similar to the earlier mentioned parity-bit experiment by Bakker in which the agent has both an RNN and an FNN at its disposal to solve the T-maze. In Bakker's experiment the agent not only learns to train both networks but also to decide which one to use.

In the system above both kinds of network are present. We will later see that the decision of using memory or not is also an integrated part of the system and actually resides in the memory blocks themselves. When we do not use the memory content the memory blocks lie "dormant" and the system functions solely on the feed-forward part.

We seek to approximate the Advantage value for all actions of each state. An obvious approach is to have a separate network for each action with the state as sensor input. Another approach is to let the action be part of the input to the system and thereby having just one network. As is discussed in [Nielsen & Hansen, 05] and [Bakker, 04], this latter design is not a good idea.

Instead of 4 distinct networks (one pr. action) I integrate these into one network with four output neurons. Each output neuron will have a dedicated hidden layer consisting of three neurons. These hidden layers all have access to the input layer. At this point, the system is still similar to four separate networks. Now, we connect the system with a group of memory blocks which act as memory for all output neurons.

All output and hidden units receive input from these memory blocks. The memory blocks themselves receive input from the all hidden units and the input layer as well as other memory blocks.

The rules for the feed-forward part is as follows. net_j is the input from the network to neuron j and y^j is the activation value.

$$\begin{aligned} y^i(t) &= \text{set externally} && (6.1)\\ net_h(t) &= \sum_{j=i,c} w_{hj} y^j(t) && (6.2)\\ y^h(t) &= f(net_h(t)) && (6.3)\\ y^o(t) = net_o(t) &= \sum_{j=h,c} w_{oj} y^j(t) && (6.4) \end{aligned}$$

Here, the activation of the input neurons is acquired from an external source. The net input of each hidden neuron, net_h, is the sum of all weights of ingoing edges (from input neurons and cells) multiplied with the activation the source neuron of each edge. The activation of the hidden neurons is simply a squashing of this input with f but the activation of the output neuron is just the identity function.

6.3 Elements of LSTM, The Forward Pass

6.3.1 The Cell, Center of Memory

The central part of the memory block is the **Cell** which is responsible for the recurrent connection. It is essentially a neuron (or neuron-like structure) with input from the system and output back to it.

A memory block can have multiple cells and a system can have more memory blocks each with its own set of internal cells. I base the description on a memory block with just one cell.

The CEC, the Recurrent Connection

At the core of the cell is the **CEC**, (Cyclic Error Carousel) which has a connection directly to itself. The connection has an edge weight of 1 which will never be altered. Therefore, it transfers the output of the CEC unchanged to itself to be used at the next time step. This connection is not considered part of the input to the cell, as it would with a normal RNN. This is the only place in LSTM with recurrent connections.

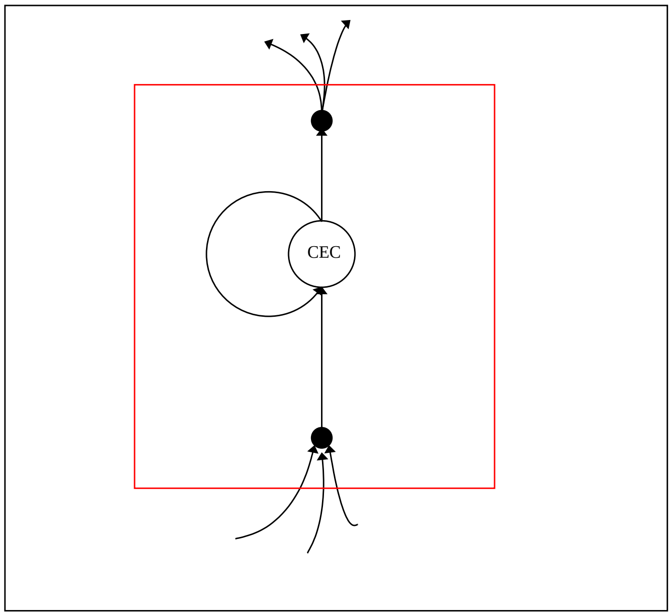

Figure 6.2: The cell is the central part of the memory block and is denoted as the red square. It has both input and output connections. At its core is the CEC which has a recurrent/feed-back connection to itself.

The content of the CEC at a given time, t, will also be referred to as the **cell state** (or $state_c$) meaning the current activation of the CEC. The cell state is computed by using the input from the network as well as the cell state one time step back at $t-1$.

The CEC has no activation function (well, identity), and is just a storage for the net input and former activation. The stored state is not considered part of the input to the CEC and especially not the cell. It should merely be considered a "real-valued variable". If the CEC receives input at some point this input will cycle unchanged forever (unless new input is presented).

This eliminates one of the problems mentioned previously. If the edge weight is different from 1, it will continually alter the output of the CEC leading to a more complex fluctuation in activation, especially if the edge became negative. If the edge weight was numerically very large (or grew to be), it would dominate the net input and make the cell state diverge.

By fixing the CEC weight at 1 we eliminate this source of trouble and are left to deal only with input to the CEC.

The squashing functions,
controling the magnitude of the cell's connections

In order to keep the input to the CEC at a manageable magnitude, the input to the cell, net_c, is squashed with the sigmoidal function, g, in the range $[-2, 2]$. Similarly, the output of the cell to the system is squashed with another sigmoidal function, h, in the range $[-1, 1]$. This is the original formulation by [Schmidhuber & Hochreiter, 95]. Gers [Gers,01] notes that no empirical evidence supports the use of the output squashing and he eliminates it in his dissertation. Bakker uses it, though, and to comply with his results I do too. It should be noted, however, that output squashing might be an unnecessary operation.

Note that the input to the CEC will be roughly of the same magnitude as the stored cell state, since the recurrent connection is 1.

6.3.2 The Ingate, Keeps Unwanted Input out of the Cell

Next, I will target the problem of the constant flow of data from the network and into the CEC by introducing the concept of the **ingate**, see figure 6.3.

All net input from the system is passed through the ingate before it reaches the CEC. If the gate is closed no input makes it past and the cell state will just recycle to the same value of the last time step.

The gate itself is a single sigmoidal neuron whose activation, $y^i(t)$, is multiplied the squashed input to the cell. This can be written as

$$state_c(t) = state_c(t-1) + g(net_c(t))y^i(t)$$

Here the current cell state is the sum of the old cell state with the squashed input, $g(net_c(t))$, multiplied with the activation of the ingate.

If the gate activation is close to 1, I refer to it as **open** and the squashed cell input is lead untouched into the CEC. If the gate activation is close to 0 the cell input is multiplied with 0 and disappears. I refer to this as the gate being **closed** and in this case the cell state will just recycle.

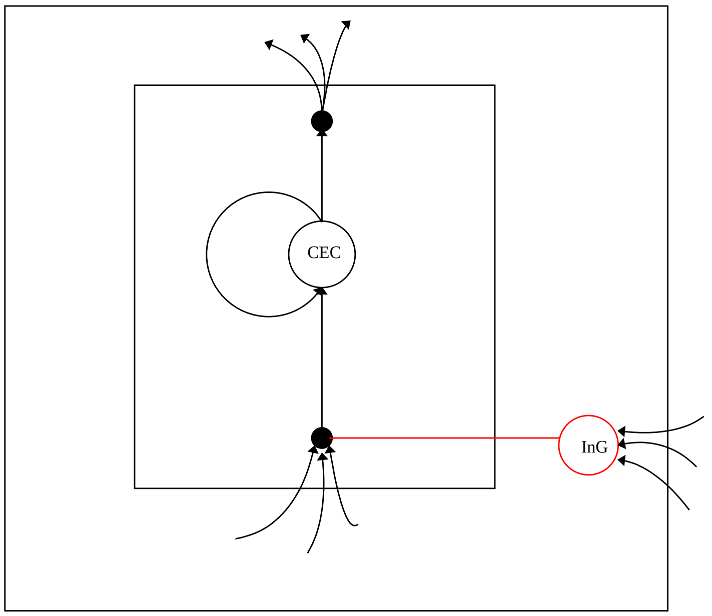

Figure 6.3: The ingate is part of the memory block but lies externally to the cell. Its activation (in range [0,1]) is multiplied the cell input before this has the chance to reach the CEC. The ingate is denoted in red.

The ingate is, in essence, a tiny FNN. So the decision to open or close the gate (or keeping it partly open) is based on the current input vector. This builds upon the intuition that the current perception of the environment should decide whether or not “something” should be “memorized”. If the environment state is “interesting” the input gate is activated so that input can flow into the cell and the CEC.

The ingate also receives input from all memory blocks and therefore it is possible to open it based on memory. These memory based connections are of lesser use though, at least in the T-maze problem. The reason is that the T-maze has very clear indication of “events”, the road sign and the junction. This will make the system utilize the direct connections to activate the ingate (and all other gates as well).

In summary the equations concerning the flow into the cell are:

$$net_{iG}(t) = \sum_{j=i,h,c} w_{iGj} y^j(t) \tag{6.5}$$

$$y^{iG}(t) = f(net_{iG}(t)) \tag{6.6}$$

$$net_c(t) = \sum_{j=i,h,c} w_{cj} y^j(t) \tag{6.7}$$

$$state_c(0) = 0 \tag{6.8}$$

$$state_c(t) = state_c(t-1) + g(net_c(t)) * y^{iG} \tag{6.9}$$

$$\tag{6.10}$$

6.3.3 The Outgate, Keeps Unwanted Output inside the Cell

With ingates we have dealt with a significant part of the problem of noisy data building up in the recurrent connection. But the data in the CEC can in most situations be considered noise to the rest of the system. The next gate is equally important to the system as it keeps the recurrent connection from "polluting" the network.

The **outgate** is similar in design and functionality to the ingate. It is located after the cell and governs the release of output to the system (see 6.4) by multiplying $h(state_c)$ with its activation.

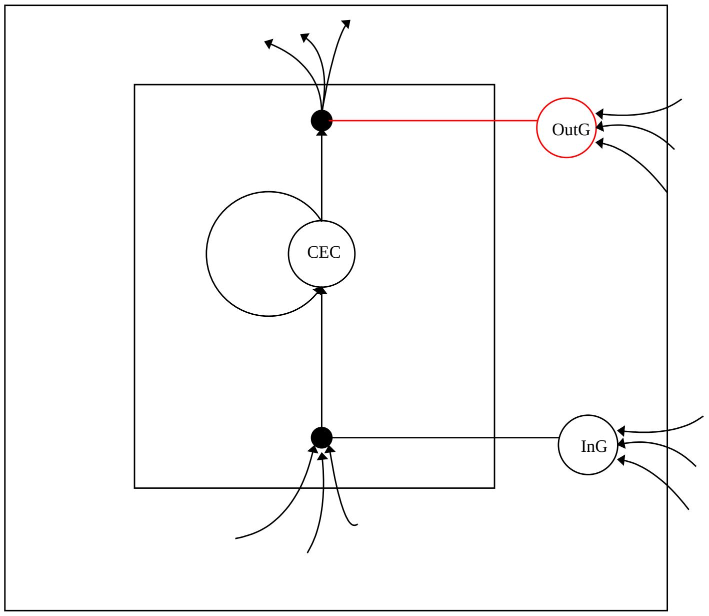

Figure 6.4: The outgate governs the release of output from the cell in a similar multiplicative fashion as the ingate. It is denoted in red

The equations governing the flow out of the cell are:

$$net_{oG}(t) = \sum_{j=i,h,c} w_{oGj} y^j(t) \tag{6.11}$$

$$y^{oG}(t) = f(net_{oG}(t)) \tag{6.12}$$

$$y^c(t) = h(state_c(t)) * y^{oG} \tag{6.13}$$

6.3.4 Forget Gates, Resetting the Cell State

Once a cell has acquired information through its ingate this information is stuck forever in the cell state. One could imagine that a similar input with the opposite sign would negate the cell state, but this just do not happen in a real system.

The reason is the structure of the network. It is very dependent on the input vector so to compute a negated input to the cell will require a specific input state which at the same time also opens the ingate to let the negation in. This is further complicated by the fact that the exact contents of the cell state is based on some state (perhaps several) further back in time. Knowledge about this exact activation is necessary to compute a negation.

After a cell state has been used it is not possible to re-use this memory block later in the episode. Traditional LSTM would usually learn to shut this memory block down by closing the outgate forever.

This is a severe resource problem. Not only would we be interested in using fewer blocks (a space concern) but it is wasteful to train a memory block to solve a given, complicated task and then throw it away just because the cell state is obsolete. The hard part is finding the edge weights and these should be reused if at all possible.

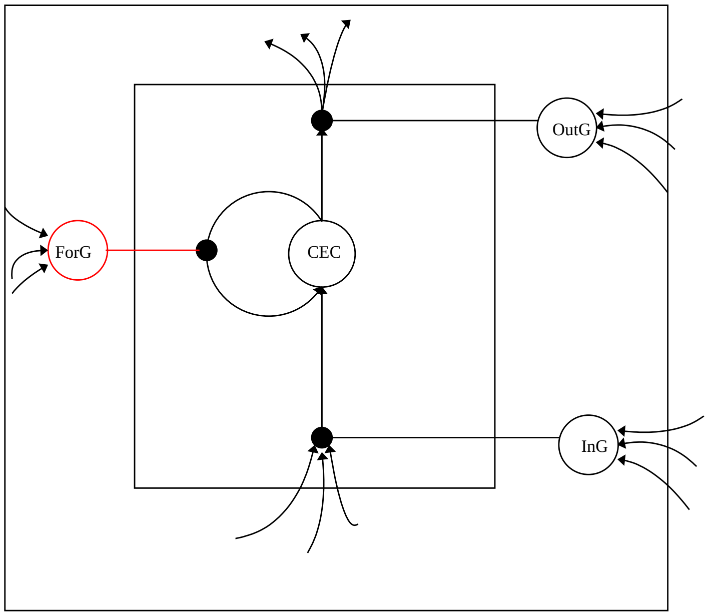

Figure 6.5: The forget gate is similar in design as the other gates. Its target is the recurrent connection in the CEC which it zeroes if necessary. In this way the system can choose to reset itself.

Gers solution to this is to introduce a **forget gate** which is a multiplicative gate similar to the two previously mentioned but its target is the recurrent connection in the CEC, see picture 6.5.

The forget gate is multiplicative like the others and alters the update equation for the cell state (6.9)

$$net_{fG}(t) = \sum_{j=i,h,c} w_{fGj} y^j(t) \tag{6.14}$$

$$y^{fG}(t) = f(net_{fG}(t)) \tag{6.15}$$

$$state_c(t) = state_c(t-1) * y^{fG} + g(net_c(t)) * y^{iG} \tag{6.16}$$

Notice how the old cell state is “gated” so that it no longer recycles freely.

Gers implemented forget gates to be used on a series of different problems. The forget gate enhancement proved to be a significant improvement on the problems requiring “forgetting” compared to traditional LSTM.

On “regular” LSTM-problems the forget gate-system performed as well as traditional LSTM suggesting that the added gate did not bring disturbance into the construction.

While the other gates usually is initialized close to 0 prior to learning (so no input and output enters or exits the memory block) the forget gate is usually initialized to a value near 1. In this way the cell state cycles uninterrupted and only later weight changes will make the forget gate come into action[1]. Essentially, “it learns to forget”.

6.3.5 Some Additional Explanations

The above description closely mirrors the descriptions found in [Bakker, 04], [Gers,01] and [Schmidhuber & Hochreiter, 95]. To enhance the intuition and facilitate understanding of the learning phase in the next section I will here touch on a few key points.

Bear in mind, however, that these are strictly to promote intuition of the construction. The ultimate “truth” lies in the equations governing the system.

Memory blocks are just an abstraction

The memory block has no functionality in itself. Everything is carried out among its internal parts, cells and gates, without any need for supervision. It is purely an abstract “wrapping” around some neurons (cells, gates) which happen to share an unorthodox relationship.

Memory blocks have no input and no output and all internal neurons have direct connections to the outside. Being a member of a memory block only decides to which in- and outgate a cell belongs.

The recurrent connection is not an edge

As already noted, the internal recurrent connection should not be considered an edge. It does not transfer output from the cell to the input but rather cycles the internal cell state. Also, it does not change during backpropagation. It should be regarded as a real-valued variable holding the cell state of the last time step. In a similar fashion the CEC should not be considered a neuron but rather an unusually complex activation function for the cell.

Cells are neurons

Cells should from the outside be considered neurons no different than any other (even though their interior is much more complex). Other neurons will “request” an activation from the cell like they do from all other cells

[1]Initialization of gates as closed is most easily made with a somewhat large negative connection to the bias neuron (and vice versa). Subsequently, learning then reduces this weight to an acceptable level.

and will similarly request an error during backpropagation. The backpropagated error will be zero, though, (see next section) but in essence they are indistinguishable from other neurons.

Gates are similar to output neurons
Gates have much in common with output neurons. Output neurons are "read" by something external to the system and posses no outgoing edges.

Similarly, gates are "read" by the cells with which they cooperate and this value is multiplied with the squashed input. There is no edge from the gates to the cells.

As error to the output neurons are computed externally, so is error to gates computed "externally" in the memory block. This is somewhat complicated procedure but happens strictly inside the memory block.

6.4 The Backward Pass, Learning in LSTM

I will derive the backpropagation algorithm as it is done in most textbooks. The LSTM part will mirror the derivations in my references though they are tailored to my specific network. The technique can handle almost any feed-forward architectures with numerous memory blocks.

6.4.1 Error Function

We wish to change the network to approximate advantage values better. We do this by changing parameters of the system and the only parameters we (are allowed to) change are the edge weights. This general idea is similar to regular backpropagation.

We consider an error function of the output of the network and the target value. With N being the number of samples (input from time steps) an often used error function is

$$E_{total} = \sum_{t=0}^{N} (\bar{T}(t) - \bar{Y}^o(t))^2 \tag{6.17}$$

Where $\bar{T}(t)$ is the target vector at time step t and $\bar{Y}^o(t)$ the output vector. The goal is to minimize this function. Instead of minimizing over all samples a common approach is to focus on one sample at a time. That is to minimize

$$E_{(t)} = (\bar{T}(t) - \bar{Y}^o(t))^2 \tag{6.18}$$

at each time step. Even though this is different from minimizing (6.17) it usually works well if we make only small changes to the system at each time step. This approach is more incremental in nature and is therefore appropriate for our system.

As a note, I only update the output neuron corresponding to the action whose advantage value which is sought updated. The remaining actions are simply left untouched (in theory set to zero error), and hidden neurons and memory blocks receive no error from these. It is, after all, intended to function as four separate function approximators in one. The error function becomes

$$E = (T(t) - y^o(t))^2 \tag{6.19}$$

where it is assumed that we consider the output neuron corresponding to the relevant action. I have removed the vector notation to denote that $T(t)$ and $y^o(t)$ are now real numbers corresponding to the relevant action.

The above error function is the most commonly used in NN literature. Here, I will use the squared TD-error instead, as this error more captures the essence of the system. The approach is to compute weight changes Δw which are added to the edge weights at the end of the learning phase.

$$\Delta w_{ij} = \alpha E^{TD}(t) e_{ij}(t) \tag{6.20}$$

where for each edge I compute changes via an eligability trace

$$\begin{aligned} e_{ij}(t) &= \gamma\Lambda(t)e_{ij}(t-1) + \frac{\partial A(s(t),a(t))}{\partial w_{ij}} && (6.21)\\ &= \gamma\Lambda(t)e_{ij}(t-1) + \frac{\partial y_o(t)}{\partial w_{ij}} && (6.22)\\ \Lambda(t) &= \begin{cases} \lambda & a(t) = \arg\max_a A(s(t),a(t)) \\ 0 & \text{otherwise} \end{cases} && (6.23)\end{aligned}$$

We assume that the (old) eligability trace is stored in the edge or similar from the last time step and the error is computed externally. That leaves only the derivative $\frac{\partial Act_o(t)}{\partial w_{ij}}$ to be computed for each edge. The way to do this is to apply the chain rule repeatedly with some special care taken for the LSTM part.

An important point is that *no error leaves the memory blocks.* This is to ensure that the effect of the recurrent connections do not "escape" the containment and into the system. The error signal from cells, but also gates, are truncated to zero when leaving the memory block.

6.4.2 Edge Updates

Output and Hidden Neurons

For edges to output neurons we have

$$\frac{\partial y^o(t)}{\partial w_{oj}} = w_{oj} y^j(t) \tag{6.24}$$

This is (implicitly) multiplied with the derivative of the activation function $f'(net_o)$ which in this case is a 1. For edges to hidden neurons we have

$$\begin{aligned} \frac{\partial y^o}{w_{hl}} &= (w_{oh} + \sum_{k=c,iG,oG,fG} \frac{\partial y^o}{w_{kh}}) f'(net_h) y^j && (6.25)\\ &= w_{oh} f'(net_h) y^j && (6.26)\end{aligned}$$

Here, I have added the connections from hidden units to the different memory block units. These are all truncated to zero and therefore disappear in the final equation. I have included it for completeness as the connections to theses units exist. As they always have zero error, they will not figure in the following.

Outgates

The outgate is the simplest of the LSTM neurons with regard to updates. For connections to an output neuron we have

$$\begin{aligned}\frac{\partial y^o}{w_{oGj}} &= \sum_c h(state_c(t))(w_{oc} + \sum_h w_{oGh} f'(net_h) y^j) \\ &\cdot \quad f'_{oG}(net_{oG}(t)) y^j \end{aligned} \tag{6.27}$$

In the first summation I sum over all cells in the block. While we only use one cell pr. block in this thesis, it is an important point that the update of the outgate (and similarly for the ingate and forget gate) is based on all the cells to which they are connected.

Notice how the outgoing "connection" from the outgate is connected to the rest of the system through the cells. It is therefore the cumulative error of all cells of the block which is used as error for the outgate. As this only depends on the current value of the cell states we notice an update similar to truncated BPTT where the update is truncated one time step back.

Cells

For connections to a cell we have

$$\begin{aligned}\frac{\partial y^o}{w_{cj}} &= \Big(w_{oc} + \sum_h w_{oGh} w_{hc} f'(net_h)\Big) \\ &\cdot \quad y^{oG} h'(state_c(t)) \frac{\partial state_c}{\partial w_{cj}} \end{aligned} \tag{6.28}$$

The last derivative $\frac{\partial state_c}{\partial w_{cj}}$ is the information that is used in the RTRL-based update. It is computed with

$$\begin{aligned}\frac{\partial state_c(t)}{\partial w_{cj}} &= \frac{\partial state_c(t-1)}{\partial w_{cj}} y^{fG}(t) \\ &+ \quad g'(net_c) y^{iG} y^j \end{aligned} \tag{6.29}$$

Notice here the implicit p^k_{ij} computation as described under the RTRL section, see (5.5).

Input gates

For connections to input gates we have

$$\begin{aligned}\frac{\partial y^o}{w_{iGj}} &= \sum_c \Big(w_{oc} + \sum_h w_{oh} w_{hc} f'(net_h)\Big) \\ &\cdot \quad y^{oG} h'(state_c) \frac{\partial state_c}{\partial w_{iGj}} \qquad (6.30)\end{aligned}$$

here, $\frac{\partial state_c}{\partial w_{iGj}}$ is derived as

$$\begin{aligned}\frac{\partial state_c(t)}{\partial w_{iGj}} &= \frac{\partial state_c(t-1)}{\partial w_{iGj}} y^{fG}(t) \\ &+ \quad g(net_c) f'(net_{iG}) y^j \qquad (6.31)\end{aligned}$$

Forget gates

Finally, for connections to forget gates we have

$$\frac{\partial y^o}{w_{fGj}} = \sum_c \Big(w_{oc} + \sum_h w_{oh} w_{hc} f'(net_h)\Big) \qquad (6.32)$$

$$\cdot \quad y^{oG} h'(state_c) \frac{\partial state_c}{\partial w_{fGj}} \qquad (6.33)$$

and $\frac{\partial state_c}{\partial w_{fGj}}$ is computed as

$$\frac{\partial state_c(t)}{\partial w_{fGj}} = \frac{\partial state_c(t-1)}{\partial w_{fGj}} y^{fG}(t) \qquad (6.34)$$

$$+ \quad state_c(t-1) f'(net_{fG}) y^j \qquad (6.35)$$

This concludes the derivation of the backpropagation algorithm for the LSTM network I use to compute advantage(λ) values. I present the algorithm in pseudo code in appendix A. It is based on the pseudo code in [Gers,01] but I have altered it somewhat and removed some superfluous loopings.

6.5 Summary and Reflections

In this chapter I have introduced a very special kind of recurrent network called Long Short Term Memory. It is based on a regular RNN but imposes a series of "containment measures" on the problematic recurrent connections. These "containers" are called memory blocks.

The cell is the neuron which has a recurrent connection to itself. It is "embedded" in the memory block. Access to the cells is severely limited by the gates which come in three flavors: The ingate, outgate and forget gate. The ingate governs whether input should flow into the cell or not. The outgate governs whether output should flow out from the cell and into the system. Also, it controls how much error should be let in the cell during back-propagation. The forget gate has the special task of zeroing the recurrent connection in the cell. It also at the same time eliminates the derivatives for the cell so the agent not only "forgets what it has learned" but also "forgets how to learn" temporarily.

The name LSTM has an explanation. The system has several features which mirror short-term memory, as it is seen in human beings. It is easy to store and retrieve information which can be done immediately at the current time step. Also, the information is fairly well-protected if not needed. The part about it being "long term" stems from the results that LSTM can breach a time lag of at a thousand time steps, see [Schmidhuber & Hochreiter, 95]. Below I comment on some of the specific features.

6.5.1 Comments and Perspectives on LSTM and POMDP

LSTM was not developed to POMDP or RL in any particular way. Rather, it is a supervised learning technique which originally was used on temporal dependencies in time series data.

As RL is an unsupervised technique we should not expect an integration with LSTM to produce systems which can perform as impressively as in the supervised case. The environment will simply make the training data too noisy for LSTM to reach such heights. Notice that we do not try to predict the reward from the environment with the LSTM-network. We try to predict our "belief" of the environment (or predict our prediction), namely the Advantage function.

The data stream is much more noisy than it would be in a supervised learning setting as is based on the agents actions which might differ much. In the beginning the agent will explore a lot and sample the environment through many episodes. The temporal correlation is still there but it is much harder to spot.

It is important that the agent at some point achieves a stabilized performance so that the temporal dependency becomes more clear. In the T-maze problem the agent must not only learn to walk to the junction but also to do this rather directly. It is unsatisfactory if it hits many walls in the corridor, walks a lot back and forth, etc.

This does not mean that LSTM cannot handle noise in itself[2]. Bakker made successful experiments on the T-maze problem where the sensor input in the corridor was subject to gaussian noise. But if the agent, for example, was forced to make 50% random actions in each state the resulting path of samples would be difficult to use.

[2]RL is actually not quite pure unsupervised learning as the designer chooses a reward function for the problem beforehand. Still, we are far away from the case where each action has a label denoting the correct choice. RL should be thought of as unsupervised learning even though this is not strictly so.

Supervised Learning with a Changing Target

When speaking of noise it is should be noted that the target function, T, for the LSTM-network (A function of the time or the environmental state, depending on viewpoint) is *not* fixed as it would be in supervised learning. It is changing over time according to what rewards the agent experiences and how much exploration is taking place and so on. It is actually partly dependent on the Advantage system which we try to approximate!

It will not stabilize until after a significant number of rewards has been sampled. Too much trust should therefore not be placed in the target value, especially not in the beginning or in unexplored areas. As the LSTM network is only changing slowly this does not have the problematic impact on learning it could have.

The Gates are "Leaking"

Notice that the choice of activation function, $f(x) = \frac{1}{1+e^{-x}}$, only makes the gates completely open at $x = +\infty$ or completely closed at $x = -\infty$. As we never reach these value we should expect the gates to always be "leaking" somewhat.

This has not been addressed in any literature I have found and it might be expected that this has a small influence in the short run (thousands of consecutive steps). Still, the system will not remain stable, especially not in continuous tasks. I do not know the extent of this problem but it could be solved with some kind of thresholding if needed. Indeed, the gates may also open, perhaps partially, due to spurious happenings in the environment.

Automatic resetting of the system

The resetting of the state (*not* the edges) of a regular FNN is fairly unexciting. The network is reset at every time step and any activation a neuron might have is overwritten with a new value.

In RNN resetting the state is another much more important matter as the recurrent connection needs to be reset eventually. Gers remarks that *any* RNN technique would need to address this problem. This is not a flaw with RNN but rather a consequence of the physical world. Very little information is important enough to be kept forever and if it were, it would probably be more reasonable to direct it to a more permanent storage, like edge weights, in the first place.

The reason that resetting an RNN has not been more discussed is, according to Gers, that few techniques for training RNN have been so successful that the problem has become relevant. That is the case with LSTM.

Resetting is usually done externally. If not done explicitly it is at least done every time the program is restarted. The problem is to figure out when to reset the state. It is easy to have a designer sit and push the reset button when necessary. It is more difficult for the system itself to learn when a task is completed.

Gers addition to LSTM makes the system able to decide internally when it is time to reset its memory. This operation is learned as well as all the other aspects of the problem which makes it a rather strong property.

Partly it makes way for solving “hidden” tasks where we do not know if resetting is needed at some point. Partly it is one less thing for the designer to spend his time on. As noted in the introduction, it is one of the key points to remove work from the designer.

Storage in a storage-less dynamic system

I have remarked that the cell state should be thought of as a real-valued variable. In a dynamic system such as ours there is no such thing as a “storage” or stable place to put variables for later use. One of the few ways for a variable to be “kept” for later use, perhaps the only way, is to put it into an eternal looping in the system. A looping which for each new time step refreshes the value so that in can be used if necessary.

Whether used or not, it will be kept for yet another cycle. The forget gate plays the essential role of updating the variable. The resetting makes it possible to assign it a new value.

Part II

Experiments

Chapter 7

Empirical studies of RL/LSTM

In this chapter I present a rather thorough experimental examination of the RL/LSTM technique with a basis in the reported results of Bakker in his dissertation [Bakker, 04]. I reproduce one of his main results for our T-maze problem, but mainly I investigate some of the key parameters in the system and reflect on the impact they have on performance.

The experimental results are grouped according the specific facet I explore. These facets are

- The κ-parameter in Advantage learning.
- The λ-parameter Advantage(λ)-learning.
- Bakker's directed exploration method.
- The number of memory blocks and their initial gate bias values.

I finish the chapter with a note on RL/LSTM's ability to handle more choices than just the 2 in the T-maze. This is explored with another artificial maze, the 8-maze, which is of my own design and has 7 escaping corridors at the junction. I show that RL/LSTM can handle such tasks with higher complexity in the temporal problem.

7.1 Bakker's Experiments

The part of Bakker's thesis concerning the T-maze scenario is summarized below. The full coverage can be found in [Bakker, 04].

Bakker uses the T-maze described in section 3.2 and experiments with both noisy and non-noisy input. When "non-noisy" the corridor input is (1,1,0). When noisy the two ones are replaced by a uniformly distributed value in the range [0,1].

An episode is considered successful when 80% of a test batch of episodes are wins. As the action selection is at least 5% exploring Bakker notes that this would correspond to optimal performance under a greedy selection.

Environment

Main corridor of length 5 to 70. A reward of 4.0 when taking the correct action at the junction. A penalty of -0.1 when taking the wrong one. Also -0.1 when attempting to move into walls.

Agent

The network design is similar to the one described in section6.2.

Nr. of memory blocks	3 with one cell in each block
Forget gates	All memory blocks use forget gates
Weight initialization, except bias to gates	Randomly in the interval [-0.1,0.1].
Weight initialization, bias to in/out gates	Set to -n, n is block number (n=0,1,2)
Weight initialization, bias to forget gates	Set to n, n is block number (n=0,1,2)
Learning rate in LSTM-network	$\alpha = 0.0002$.
Learning rate in TD-error estimating network	$\alpha = 0.01$
Scaling factor for directed exploration	c = 0.2
Discounting factor	$\gamma = 0.98$
Eligability trace decay	λ=0.8
Scaling factor in Advantage learning	κ=0.1

Results

- Successful completion of mazes with corridor lengths up to 70
- For a given episode a trained agent has the following properties:
  - Memory Block 1 seems to be useless. Its internal cell state grows continuously as the ingate is always open. Its outgate is always closed.
  - Memory Block 2 seems to encode the west goal. If the initial sign is "west" the ingate opens and closes afterwards. The outgate is closed except at the junction.
  - Memory Block 3 seems to function like memory block 2 except for the "east"-sign
- All forget gates are open (therefore unused) during the entire episode.

7.2 Experiments with Bakker's setup

When contemplating the experimental problem and his results some questions come to mind.

1. *is Advantage learning necessary at all?* Perhaps pure Q-learning would work just as well. If not, what is the difference?

2. Similarly, *Is Advantage(λ) learning necessary?* From classical RL it is known that eligability traces only improve convergence time and has no effect on the achieved learning. Perhaps pure Advantage learning would work as well.

3. *Is Bakker's Directed Exploration technique necessary?*. Perhaps the fancy construction is superfluous or does not do what we hoped it would.

4. *Are three memory blocks necessary?* As the decision is rather binary a one-cell-one-block design ought to be sufficient.

5. *Are the forget gates necessary?* The problem has no instances of things needing "forgetting". Also, Bakker's own results show that the forget gate remains wide open throughout the episode.

Based on elementary RL theory, our intuition would be that the answers to all these questions would be negative as all are augmentations to a basic mechanism which can work without them.

Based on the discussion in the theoretical part, especially with the inclusion of function approximation, we might believe that the first three are positive but that maybe the last two are negative.

Surprisingly, the answers are indeed all negative but some aspects, especially directed exploration and Advantage learning, prove to have a large impact on the system. Also, some aspects have an adverse effect on performance.

I explore these aspects beyond "Strictly necessity". That is whether and how much they improve or decrease learning and convergence. I cover these points:

- I experiment with the κ-parameter in Advantage-learning and show the large impact of this parameter on the behavior of the agent.

- I show experimentally that the addition of eligability traces to Advantage learning, Advantage(λ)-learning, has no or even a worsening effect on both ability to learn and convergence time. This is important and surprising as the main reason to introduce eligability traces in the first place is to *improve* convergence time.

- I show the clear advantage of using Bakker's novel directed exploration technique compared to undirected, ϵ-greedy, exploration.

- I show that having too few memory blocks has much impact on performance whereas "superfluous" memory blocks do not have a decreasing effect.
- I confirm Gers' comment in [Gers,01] that the choice of initial gate bias makes little difference.
- I reproduce Bakker's results concerning the T-maze with varying corridor length. I will show that Bakker's reported maximal corridor length of 70 is not the limit. I report successful completions of T-mazes with corridor length up to 144.

Other Parameters

Many parameters in the setup can be adjusted (The command line parameters number around 30 and this does not include several default parameters in the code) but I will only discuss the above mentioned.

Besides those at least to other points in Bakker's setup is of interest: the choice of middle layer (1 layer 3 neuron in each pr action) and the choice of one cell pr. block. The former choice is really just the "usual question of middle neurons/layers" which is well-known (but not well-solved) in feedforward literature.

The latter, though, deserves a comment. The gating decides when information flows in and out of a block. To have more cells in a block amounts to having more complicated computation taking place. Even though the cells are recurrent connections and thus in theory ought to have strong computational power, they should be considered more like inanimate storage and it is the gates which create the inanimate aspect of the block. With more cells a more complicated pattern could be "stored" in the block. The designer therefore needs to decide how complex the information he wishes to store is. In my experimental setup the choice at each junction is almost binary and I have therefore chosen to keep just a single cell in each block.

7.2.1 Experimental setup

My setup mirrors Bakker's except that I do not consider noisy input. I usually only adjust a single parameter in the experiments.

Test Runs

I work with an **execution** of the program. Each execution is initialized with random settings for edge weights and fixed values for all other parameters. After initialization the only parameters which change during learning are the edge weights.

I group executions into a batch-test called a **test-run**. A test-run consists of a 100 independent **executions** of the program. This is sufficient to give a general idea of the parameter's influence.

An episode is ended either because of **episode timeout** or because the agent escapes the maze. Episode timeout is simply a limit on the number of state transitions the agent can make. When an episode ends the agent is restarted at the beginning but all activation values, derivatives and cell states (ie. everything but edge weights) are zeroed.

Each execution is evaluated at every 1000 episodes. If an agent wins the maze in all episodes the execution is terminated. Otherwise it is terminated after a certain number of episodes, the **Execution timeout**, which usually is corridor length*5000. This is a fairly low value and Bakker reports convergence after hundreds of thousands of episodes. The reason for this low timeout is a concern for testing time as a test run easily can take many days on the complicated cases.

As the results generally are acceptable I find this sufficient even if they might be improved with more episodes. Also, I have during my work found a tendency for agents in this kind of problem: If they do not solve it quickly, they never solve it.

As an example, if an agent typically uses 12.000 episodes (+/- 5.000) to solve a task and it has spent 100.000 episode on an execution, then it will never solve it (even with several hundreds of thousands of additional episodes). Of course we never know for sure but it seems as it becomes stuck in a useless extrema in parameter space.

Success criteria

An execution is a success if the agent wins 950 episodes out of a thousand. The reason for this criteria is the minor randomness (5%)in the directed exploration. It will not be telling to demand 100% succes as just a single or few forced explorations would throw a trained agent off. The reason not to terminate at this point is to investigate if the agent achieves perfect performance.

The minimum of 5% exploring actions in the junction does not lead to losses immediately as the north and south choices are possible as well. Bakker uses an 80% success rate to denote a successful execution but even though this value may seem much lower than mine, it is still significantly above chance and the two success criteria might be close to equivalent. The 80% mark, though, might happen significantly earlier than the 95% mark which will make the convergence times different.

Point of Realization

If an agent is successful it is worth noticing when it "learned" the road sign. Learning or "realization" usually happens suddenly in RL.Once the correlation between road sign and junction is perceived the agent quickly learns to utilize it. To find this point I define the **point of realization** as the first time the agent wins in more than 540 episodes.

This might seem like a low limit to denote realization. But if one assumes that the act of winning a maze is purely random by the agent (50% chance of choosing the right direction) then the sum of all the correct decisions can be considered binomially(1000,p) distributed with p=0.5.

To statistically test whether an observation could stem from such a distribution a likelihood test would consider 541 to be dismissed on a 0.01% significance. In short, if we see 541 wins or more the agent cannot statistically be considered to be completely random in its choice of action but biased toward the correct decision.

7.2.2 Reported Results: Success Rate and Convergence Time

For every experiment two results are reported, success rate and convergence time. I also use the term **performance** more informally to denote the "usefulness" of the agent, especially with regard to these two aspects (but *not* with regard to execution time).

In every test run the agent is initialized 100 times with random edge weights. The **success rate** is the percentage of these 100 executions in which the agent learns to solve the problem.

If we just have a single win then we know that the agent was able to solve the problem. If we have a 100% success rate we know this as well but also that the agent is fairly robust as to which parts of the parameter space it is initialized in.

A more biased initialization might change an unsuccessful agent to a successful one. With at least one success we know it can be done, but with more we may consider the agent more suited to handle this kind of task.

The **convergence time** is how long it takes for the agent to solve the task, if it can be done. Generally, I present the median, the lower quantile and upper quantile. I find this more explanatory than the empirical mean and variance, as there are often several outliers blurring the picture.

If we have n successful executions and sort by convergence time then the $\frac{n}{2}$'th execution is the median. The $\frac{n}{4}$'th execution and $\frac{3n}{4}$-th execution are the lower and upper quantile respectively.

In some plots I include a "regression line" which is the line that minimizes the least squared error to all data points. This line is added as an assistance to the reader in order to clarify the data.

The use of this line does not imply a test for linear regression. This would be applicable if it was reasonable to assume a linear relation between the values on the x-axis and the y-axis and we wished to test for this.

Even though several results seem to "fall along a line" and very well might have a linear relationship I have not found it relevant to test for this nor to uncover such relationships. Still, the regression line gives a rough idea of the distribution of the results.

7.3 Experiments

7.3.1 The κ-value in Advantage learning

Consider the κ value in Advantage learning. The update is

$$A(s_t, a_t) \quad \leftarrow \quad \max_a A(s_t, a_t) + \frac{R_t^\lambda - \max_a A(s_t, a_t)}{\kappa}$$

Recall that κ scales and accentuates differences between the expected result and sampled result.

As mentioned earlier only $\kappa \in (0, 1]$ makes sense. The smaller the κ-values the greater the difference between Advantage values but it is unclear how small a "good" κ is.

I have tested with κ-values belonging to $[0.01, 0.02, \cdots, 0.99, 1.00]$. Notice that the 1.00 value would reduce to standard Q-learning and that 0.10 is Bakker's choice. I test on a maze with corridor length 5.

The agent is allowed a maximum of 80,000 episodes. This is more than the 20,000 (5,000×corridor length) I generally use but with $\kappa > 0.1$ the agent was mostly timed out and this is the reason for the increased execution timeout.

Even though we might expect the very low values to be of greatest use it is important to get a rather good impression of the impact of the parameter. Especially when it approaches 1 which is Q-learning.

Results

The success rate is depicted in figure 7.1 and the median convergence time in figure 7.2.

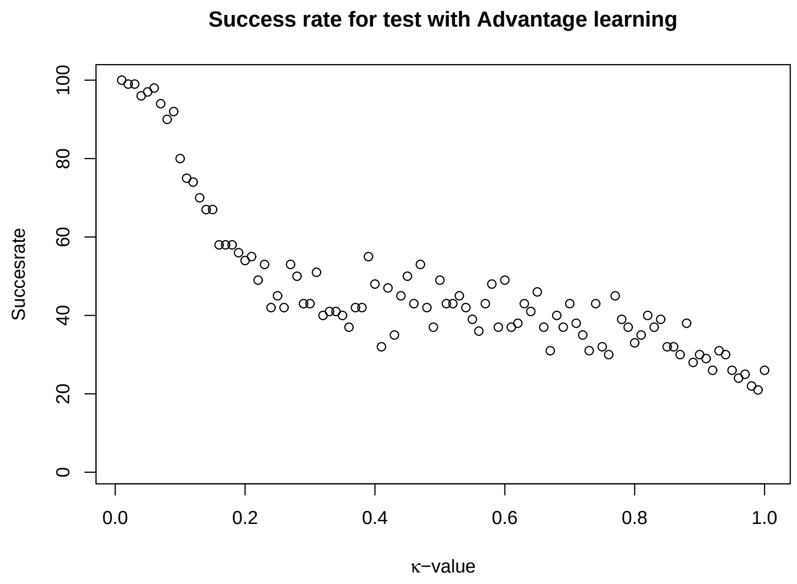

Figure 7.1: Success rate for κ-values between 0.01 and 1. Notice the significant drop as κ approaches 0.2. $\kappa = 1$ is conventional Q-learning.

Success rate

Notice that the success rate decreases significantly as κ increases. This is interesting especially as it happens so early as around $\kappa = 0.12$. Q-learning ($\kappa = 1$) is seemingly the poorest choice and the values close to zero has a significantly better impact than most other. Values larger than 0.2 perform almost identically and this might stem from the polynomial use of the parameter in the system. Low values accentuates differences much more than higher.

Convergence Time

The median convergence times are even more drastic. The small κ-values have excellent convergence times of around 4000 episodes whereas this number quickly rises many times this amount as κ grows. The seemingly linear relationship is a surprising feature.

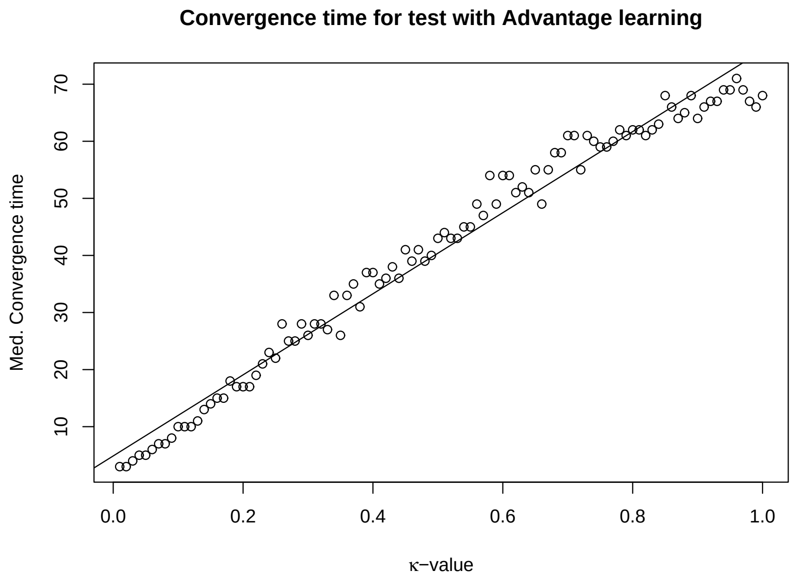

Figure 7.2: Median convergence time for κ-values between 0.01 and 1. An oddly linear relationship is prevalent. Convergence time is in thousands.

It is obvious that this problem type needs a fine-grained ability to distinguish between optimal and suboptimal choices. It seams as the Advantage values are very close but that the optimal choice is slightly larger than the other choices. When this differences is enlarged the policy will have greater chance of choosing the optimal one. As is also obvious, Q-learning is not a good choice.

7.3.2 Impact of Eligability traces

I test with λ set to values in the range $[0.00, 0.01, \cdots, 0.99, 1.00]$. A special note should be made about the values 0.00, 1.00 and 0.8. The first means that we zeroize traces. This makes the update depend only on the last time step and this is similar to original Advantage learning without eligability traces. The value of 1.00 is what is referred to as “Monte Carlo” in [Sutton & Barto,98]. This makes all prior states influence the current state. Even a value of 0.99 does not have this effect as it decays exponentially ($0.99^{10} = 0.904$, so it is still somewhat slow). Finally, the value of 0.8 is the one chosen by Bakker.

I test on mazes with corridor length of 5 and 15. As eligability traces are designed for looking "ahead" in time they are more useful with task which require many state transitions before completion. So especially the maze with the longer corridor may show interesting results as each episode involves many time steps.

λ is used in an exponential fashion in the algorithm. This makes values close to zero very similar as rewards further back have very small influence. As an example, if we choose $\lambda = 0.2$ the reward two time steps ahead will be multiplied with $0.2^2 = 0.04$. This value will hardly matter to the update and even less the ones further ahead. Similarly, values closer to 1 all tend to have a "long tail" of eligabilities back in time.

Results

The results are surprising; the introduction of eligability traces has an adverse effect on performance.

Success rate

The results are summarized in figure 7.3. The λ-eligability trace actually

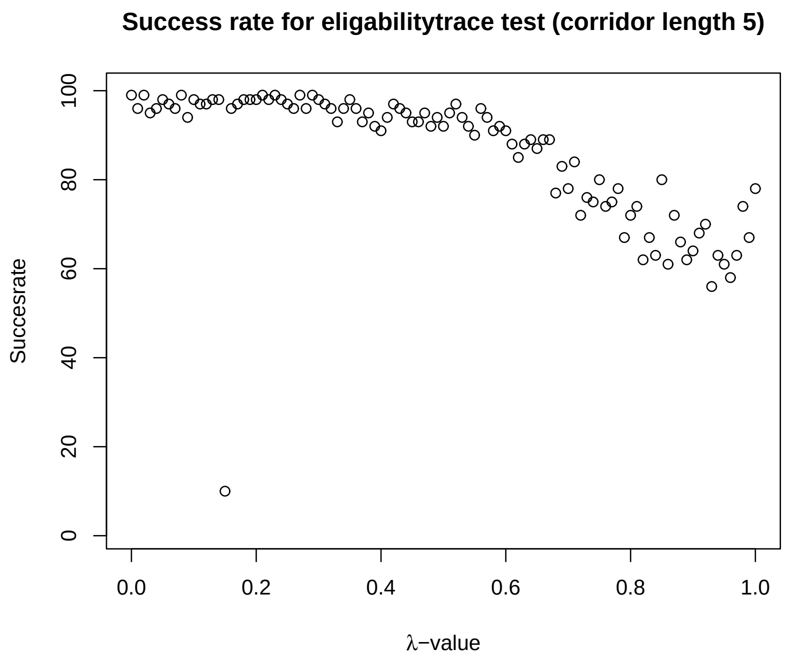

Figure 7.3: The success rate on a corridor of length 5. λ is displayed on the x-axis in intervals of 0.01. For each λ a 100 executions is made totalling 10.000 executions. There is a clear tendency that a growing λ has a worsening effect on success rate.

decreases the success rate of the agent. Perhaps the short corridor makes the problem so small that the overhead of using eligability traces plays a large role. The results, though, are not much better when the corridor length grows to 15 as displayed in figure 7.4. Here the results are much more scattered

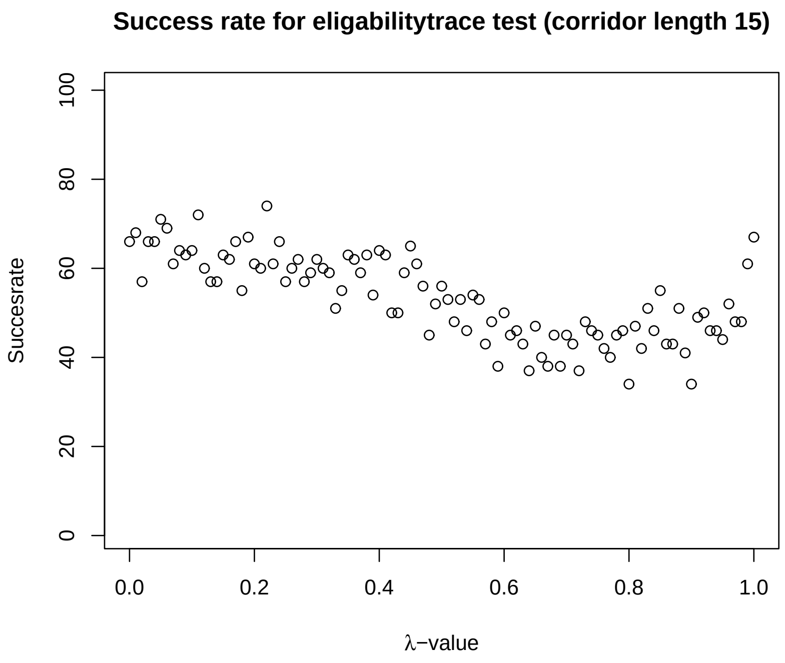

Figure 7.4: The success rate for a corridor of length 15. Again, the performance declines significantly as λ grows.

which is expected as the episode length is (at least) trippled compared to the 5-corridor maze. This gives greater variance in the exact point in time where the agent learns the task.

It still seems like the eligability trace has a small negative effect on the success rate and it clearly does not *improve* on the system.

Convergence Time

We would expect more significant results on convergence time, as this is the primary reason to use eligability traces in the first place.The median convergence times for the 5-step corridor are displayed in chart 7.5 and those for the 15-step corridor on chart 7.6.

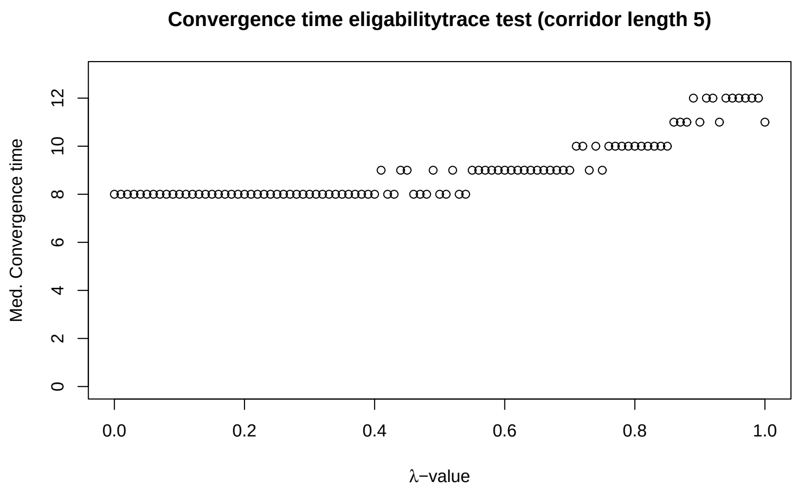

Figure 7.5: The convergence time a corridor of length 5. The convergence seem rather stable but still a significant increase as λ grows.

Here the results are even more surprising as we seem to have an *increasing* convergence time when using eligability traces in this problem.

I have included the 1st and 3rd quantile in the 15-corridor plot which gives a rough idea of the distribution of results. As the two quantiles are rather close to the median it seems that the system behaves rather stable as λ grows, as opposed to become more erratic in its performance.

It seems as if the problem has to be based on the immediate reward one time step back. Perhaps the nature of this problem makes it more problematic for the agent if all the initial (sub-optimal) exploring actions “lingers” for a longer time in the eligability traces before they disappear.

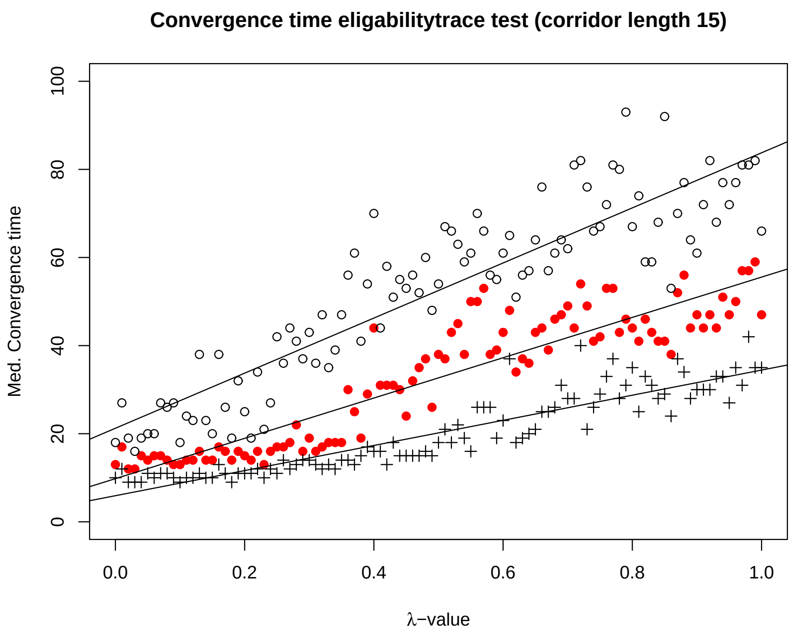

Figure 7.6: The convergence time a corridor of length 15. The charts display the 1st quartile (crosses), median (filled red circles) and 3rd quartile (open circles). Here we also see a significant increase in convergence times as λ grows.

7.3.3 Directed Exploration vs. Undirected

To get an idea of the impact of directed exploration I experiment with both directed and undirected exploration and compare the results. I test on corridors of length 5 and 10 in order to have some gap between road sign and junction.

For the undirected method I choose an ϵ-greedy algorithm with a 5% exploration. As the directed exploration is used throughout the rest of the experiments, I compare with the similar setup which is used in the corridor test described later in this chapter.

Results

In the test run with the 5-step corridor maze the agent has a success rate of 8% and the median time is 12,000 episodes (based on these 8 wins). In comparison the success rate is 79% with a median convergence rate of 9,000 episodes when using directed exploration.

The agent in the 10-step corridor maze has a success rate is 5% with a median convergence time of 21,000 episodes. With directed exploration the agent solves the task in 39 % of the executions and a median convergence of 18,000.

While I had expected worse results (initial tests were extremely discouraging) it is interesting to note that the ϵ-greedy policy can actually learn the task in some instances. It is obviously far less robust than the directed exploration and the expected convergence time seems to be higher.

Perhaps this approach could be improved with the Boltzmann distribution or with a larger ϵ which then declines and so forth. I believe such paths will not be fruitful, though, the directed exploration is in a completely different league.

7.3.4 Many and Few Memory Blocks

This experiment tries to shed light on the role of the memory blocks. It is not only a question of whether one memory block should suffice but also what initial gate bias these blocks should have. Even though one might believe any initial bias-value would be useful (as the algorithm would change it) it can throw the agent into a suboptimal hole where an acceptable performance would never emerge. I therefore change two parameters in these test, the number of blocks and the initial gate bias.

I test the problem with 1 through 7 memory blocks. Block 1 is initialized with a bias ranging from -2.5 to 2.5 in 0.5 increments. Subsequent memory blocks are initialized with increments of 1 over the prior block. I will not initialize two different blocks with the same gate bias. This will make them too much alike with the only difference in their initial random weights.

With more blocks than strictly necessary, the system might be better equiped to solve the problem at hand as it does not rely on one specific block being initialized correctly. Also there is a possibility that more blocks could generate an emergent behavior.

We might expect that a very low initial bias would be advantageous to the agent (-1 or -2, compared to the low magnitude of the weights ([-.1,.1])). The low value will close the in- and outgate in the beginning of the execution when the agent has little understanding of its surroundings. In this state it will probably not be able to infer any time dependent relations. Later, when the feed-forward part has learned the MDP-part of the problem, the focus can be on the junction and the memory part can be "activated".

Whether or not this argument is valid is uncertain and I therefore use positive values as well. Gers notes [Gers,01] that the choice of initial gate bias make little difference.

Results

As I test with two parameters, number of blocks and initial gate bias, I present the results in two graphs. One graph is indexed with blocks along the x-axis and the other with gate bias along the x-axis.

In general a 3D plot would have been more appropriate for such 3 dimensional data but the specific results obtained are more clear this way.

Success rate

Figure 7.7 displays the success rate with regard to the number of blocks used and figure 7.8 displays the success rate with regard to the initial gate bias.

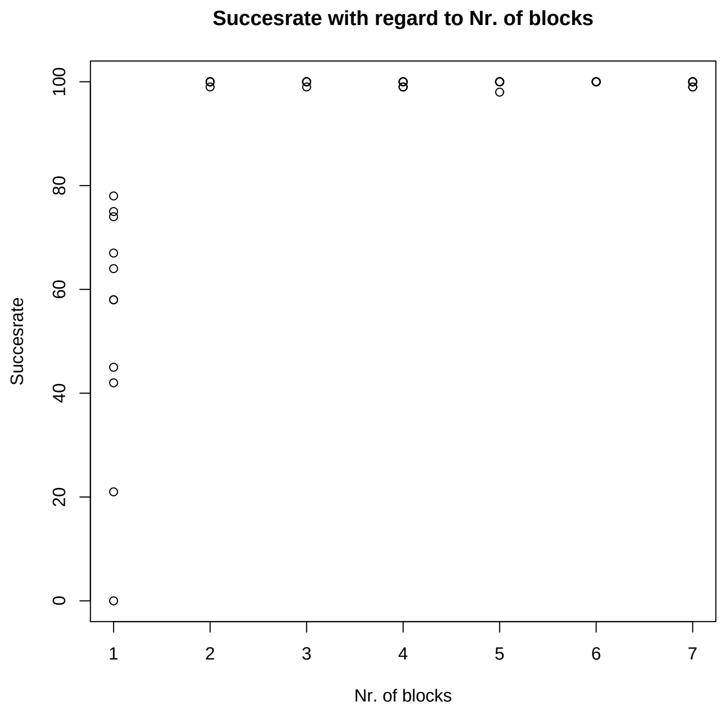

Figure 7.7: Success rate for different blocks and gate bias. Every point is the result of a test run with the block noted on the x-axis and an initial gate bias belonging to -2.5,-2.0,...,2.0,2.5. For one block the initial gate bias makes a difference but the success rate is never perfect. With two blocks or more the initial gate bias makes no difference.

These figures suggests that more memory blocks than two is unnecessary. As expected, the problem can obviously be solved with one memory block but the solution is a more fragile than with more blocks. With two blocks however, the performance is perfect regardless of initial gate bias. This suggest two things.

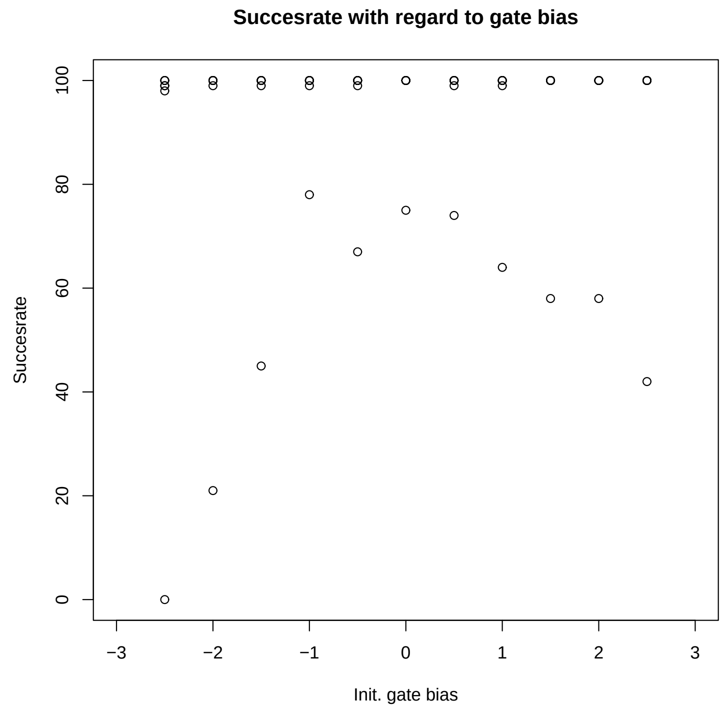

Figure 7.8: Success rate for different blocks and gate bias. Every point is the result of a test run with the initial gate bias noted on the x-axis and 1 to 7 blocks. The results with one block are the points below 90% success rate. With two blocks or more the success rate is perfect or close to.

First, the role of the initial bias is not that big. This is noted by [Gers,01] as well and is not surprising considering the many iterations. It plays a role in the 1-block system which is less suited for the task, though.

Second, the problem seems to be a "two-block"-problem. Meaning that the network finds it "convenient" to let one encode the eastern choice and the other the western. This is similar to Bakker's results mentioned earlier. With more blocks the superfluous blocks are closed (or otherwise made benign to the system, see next chapter) and it does not seem like these hurt performance.

Convergence Time

The median convergence time for two and more blocks is about a fourth of the case with one block. It is obviously a harder task to learn with one block.

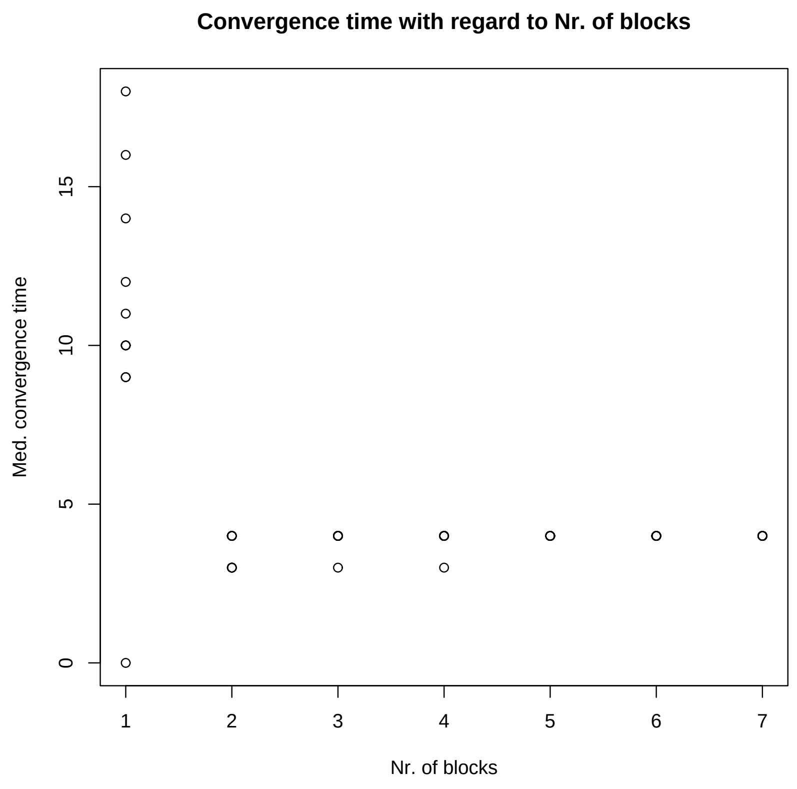

Figure 7.9: Median convergence time with regard to the number of memory blocks. With one block the convergence time is significantly longer. With more blocks, however, the convergence time is rather unaffected by the additional blocks.

With more blocks than two the convergence time is unchanged. This also suggests that the superfluous blocks do not contribute positively nor negatively to the solution. In this maze it is probably difficult to perform any better as there has to be some lower bound on how much time is needed to learn a temporal relation from scratch.

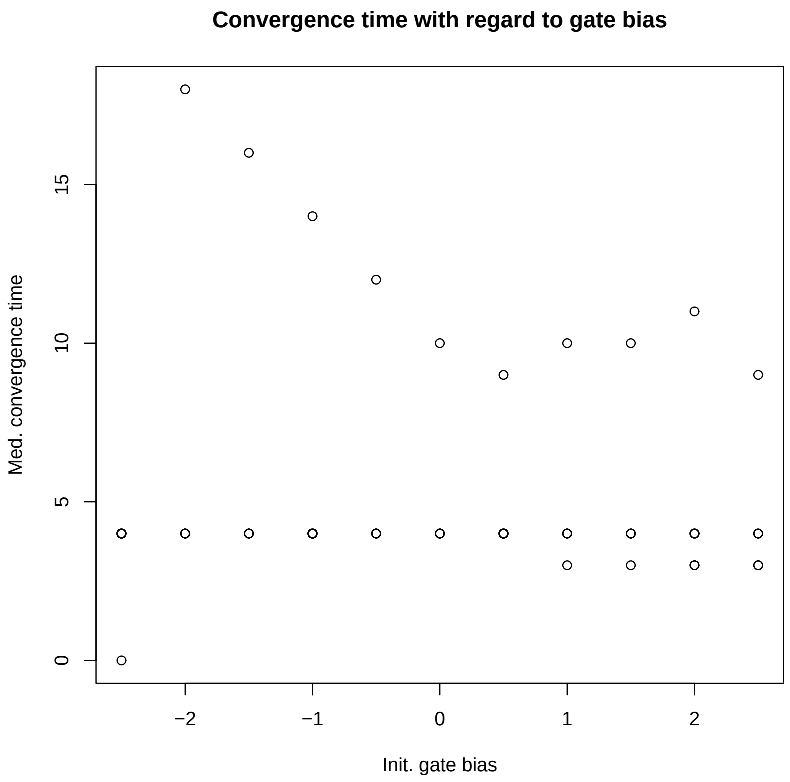

Figure 7.10: Convergence time compared with regard to initial gate bias. All convergence times above 5,000 episodes belong to the one-block system.

7.4 Reproduction of Bakker's Experiments on Corridor length

In this test I will not alter any parameters but reproduce Bakker's results on T-mazes with corridors of length 2 and upwards. Bakker reports successful completion with corridor lengths from 5 to 70.

7.4.1 Results

I have made test runs with corridor lengths from 2 to 40. From corridor length of 41 to 57 testing time proved to be a serious factor and I reduced the number of executions pr. test run to 10. This still gives some perspective on longer time lags even though the results are less certain, especially with regard to convergence time. As the last 5 smaller test runs were unsuccessful, I terminated the test.

Success rate

The success rate is displayed below.

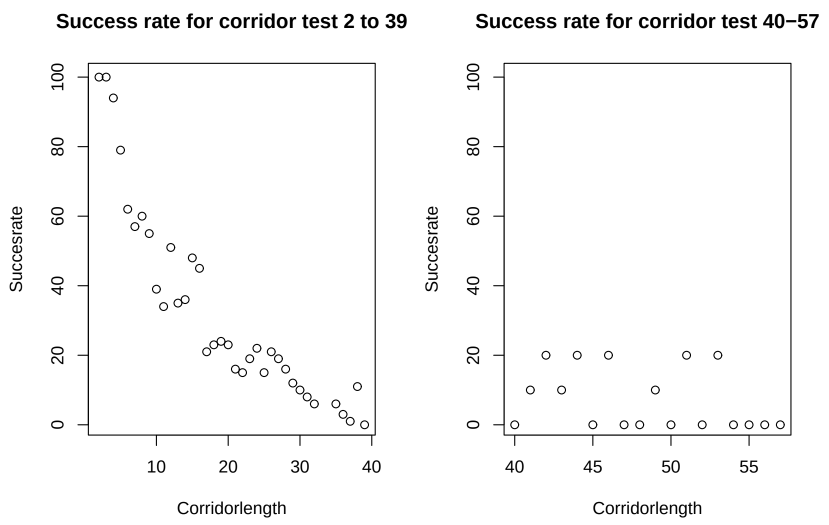

Figure 7.11: Success rate for corridor lengths 2 to 57. From length 41 to 57 I only make 10 executions pr. maze.

There is a definite decline in the success rate as the corridor grows. This is indeed expected as the longer corridors introduce longer gaps between the junction and the road sign.

Even though [Schmidhuber & Hochreiter, 95] notes examples of agents breaching a thousand time steps these are results based on supervised learning tasks. As I have noted this task is more "muddy" so this performance is acceptable and similar to Bakker's.

Even though I stopped the testing at corridor length 57, I made a single full test run (100 executions) on corridor length 70. It had a sole success after 117,000 episodes which confirms Bakker's result that the technique can solve mazes with corridor lengths up to 70.

Convergence Time

The median convergence time is shown below in figure 7.12. As expected the convergence time increases with increasing corridor length.

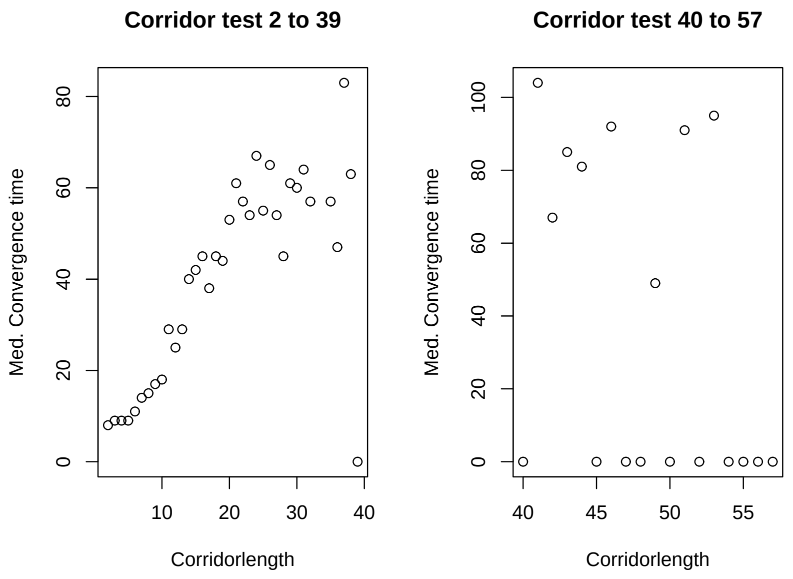

Figure 7.12: Convergence Time for corridor lengths 2 to 57. The Y-axis is in thousands of episodes. For corridor lengths over 40 the data is based on just 10 executions pr. maze instead of 100.

I have noted that RL is a "sudden" learner or that learning seems to occur in bursts. In figure 7.13 I have added the two other statistics to figure 7.12. The first is the point of realization in performance, ie. the point where the agent wins more than 540 episodes. The second is the point where perfect performance (100%) is achieved.

The points of realization are written as triangles and correspond to the circle directly above them. Note that realization generally happens shortly before success. This corresponds to usual behavior in RL where the agent suddenly learns a path to a goal even though it takes a long time to find it. The temporal feature of the problem does not change this behavior.

The perfect performances are noted with crosses and similarly correspond to the circle directly below them.These do not seem far above the point of success (95%) which seems to correspond with a difference only on the small forced exploration.

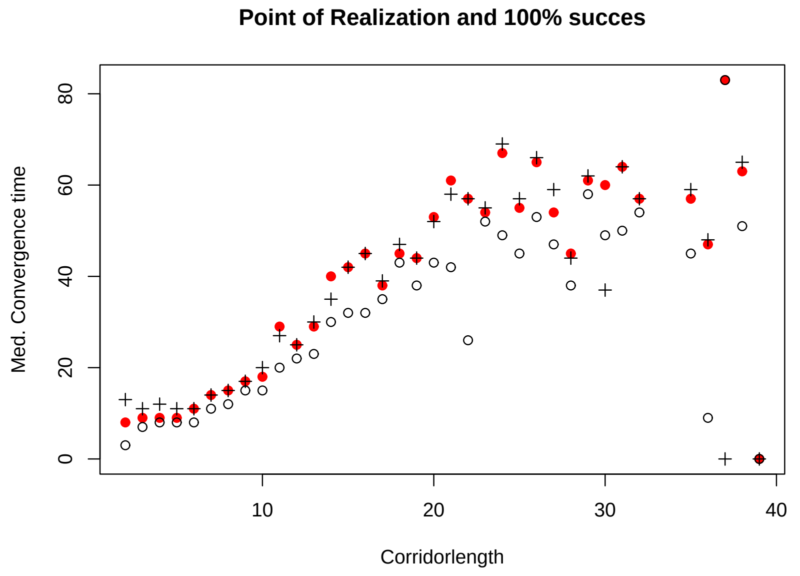

Figure 7.13: Median convergence time for test runs for corridor lengths up to 50. The filled red circles denote the median time to success (95%) and the black crosses the median time to perfect performance (100%) when this is achieved. The triangles denote the "point of realization". Notice how realization happens shortly before successful performance which again is very close to perfect performance. A few crosses are lower than the mark of success. This is because the executions where success was reached, but not perfect performance, is not included among the crosses.

7.4.2 Corridor test with improved parameters

I have shown that the κ and λ values that Bakker uses ($\kappa = 0.1$,$\lambda = 0.8$) might not be optimal for this problem. It would therefore be interesting to see the performance especially on larger corridors with a better parameter setting. I have made a small test to explore this with $\kappa = 0.01$ and $\lambda = 0.2$. These values are chosen arbitrarily in the better part of the parameter space. I test on corridor lengths of 40 and upwards but only 10 executions pr. test run.

Results

The system performed well on mazes with corridor lengths up to 144. I stopped the test at 144 in order to have time to include the data in this report, so this is probably not an upper limit. In fact, the program was processing the 145-length maze and had already produced a "win".

Figure 7.14 shows a definite improvement in success rate of the agent even though we only make 10 tries on each maze.

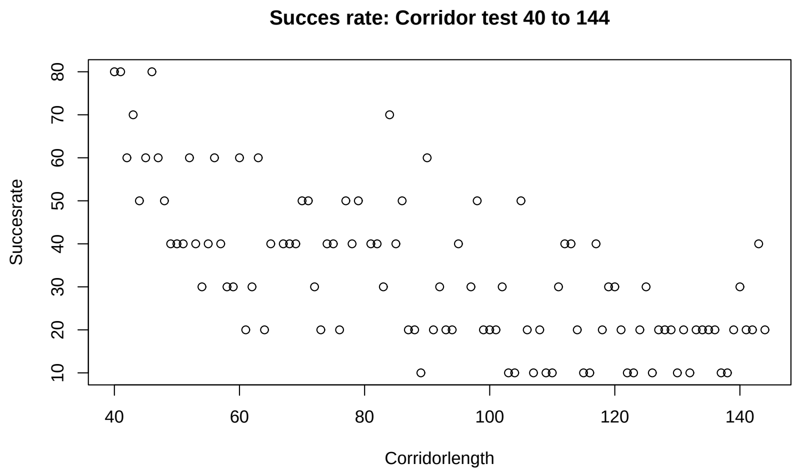

Figure 7.14: Success rate for corridor lengths 40 to 144 with more optimal parameter setting ($\kappa = 0.01$ and $\lambda = 0.2$).

The median convergence time of the test with changed κ and λ parameters is displayed in figure 7.15

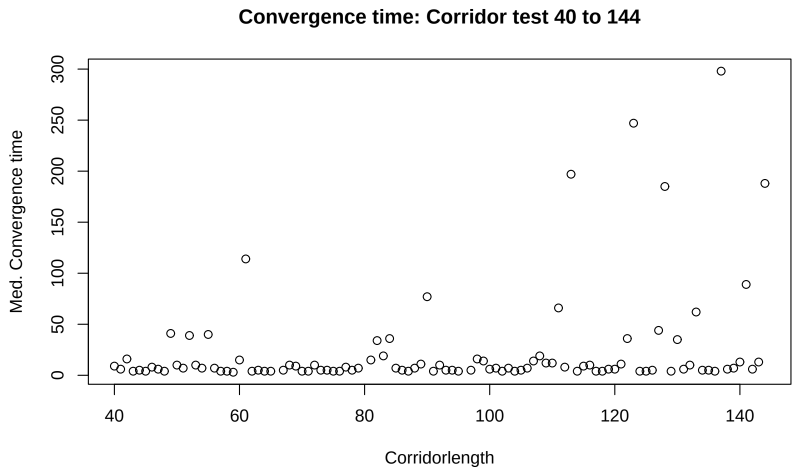

Figure 7.15: Median convergence time for the changed κ and λ. Besides a few outliers, which probably are due to the low number of tests, the convergence time is largely unaffected by corridor length. The Y-axis is in thousands of episodes.

It is surprising to see such a low median convergence time which hardly changes with the corridor length. It seems as the new choice of parameters have a great impact on the performance but more tests would be needed to confirm this with significance. Also, it seems that the agent becomes somewhat immune to the length of the corridor based on the convergence time. Still, the corridor length makes the time lag between junction and road sign longer. Notice that the agent in 2 out of 10 executions solves a maze with corridor length 144 compared to our single success in 100 with the less optimal parameter settings with maze length of 70.

7.5 Test on the 8-maze

One might fear that the earlier results are influenced by the fact that the agent only has two real possibilities at the junction. Even though it always has four actions at its disposal it will soon learn that only the eastern and western choice is relevant. Hypothetically, the agent might learn just one road sign and follow this correctly. If this road sign is absent it will never bother to learn the other one but just make the opposite action at the junction. It is important to see if the technique can cope with more information which needs to be stored.

I devised another maze depicted in figure 7.16, the purpose of this maze being to get some idea of the performance with more than just paths at the junction. In this maze the goal is placed randomly at one of the 7 “arms” and a road sign is placed at the beginning.

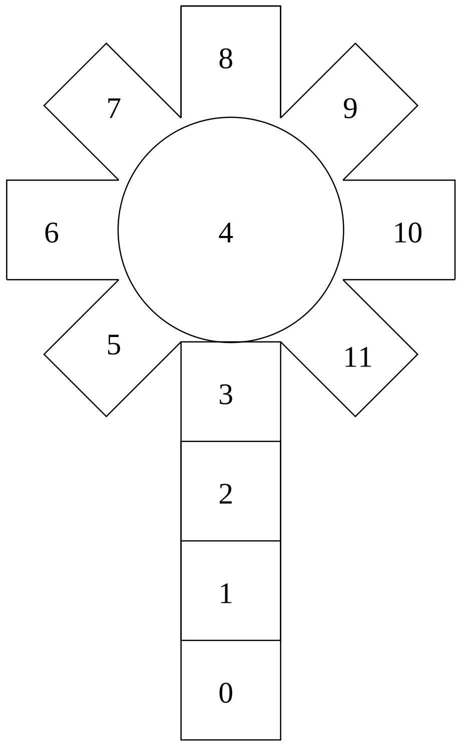

Figure 7.16: The 8-maze is a augmentation of the T-maze. Instead of a binary problem in the junction the agent now has to choose correctly between 7 different paths.

The choice of 8 actions is to approach what a real, moving robot would have at its disposal. Even though it could move forward with all kinds of speed it is generally not too limiting to restrict its actions to “move forward”, “turn left/right”, “move backwards”, “180^o-turn” and so on. Throughout my previous work [Nielsen & Hansen, 05] the real-time robot could just move forward and turn left/right and still be able to solve even the complicated problems.

I use 6 bit to encode the sensor input and gives the robot a choice between 8 actions in each state. In the corridor only north and south actions will produce state change.

I use $\kappa = 0.01$ and $\lambda = 0.2$ due to the earlier promising results. I also use 7 memory blocks and a max number of episodes of 500,000. All these parameters are educated guesses, as I have not tested this maze as thoroughly as the T-maze.

7.5.1 Results

In the limited testing I did the agent never achieves a 95% level of succes as it did on the T-maze. It is still far above chance which I illustrate in figure 7.17. Here I show to full executions starting from the first episode. The success rate stabilizes after around 90.000 episodes.

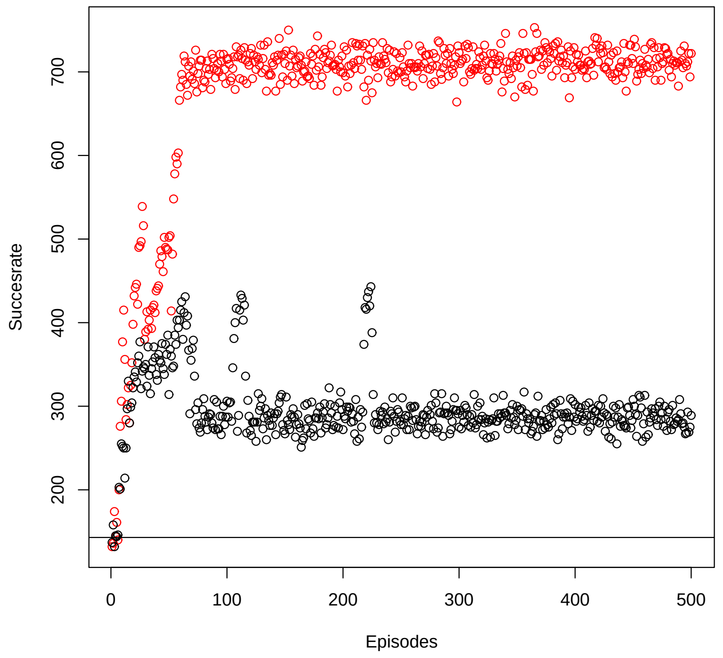

Figure 7.17: Two single executions on the 8-maze. Circles and triangles denote two different executions. Their success rate is plotted from the first episode and forward. A performance significantly above chance (14.3 % denoted by the horizontal line) is evident almost as if the agent learns part of the problem. Episodes are in thousands.

As 1 out of 7 corridors is the winning goal, it is expected that around 143 episodes out of a thousand would be correct if the agent operates at chance level. Both agents clearly learn to perform above chance and the best performance is of around 700 wins out of 1,000. This is better than just learning two or three correct corridors (the numbers suggest that about 5 out of 7 goal directions are learned).

This is amazing as it seems to learn the problem partially with a few troubled areas. A deeper study of activation fluctuations in the robot, the choice of parameters and maybe also encoding could reveal interesting aspects of this multi-goal problem type and might improve it.

Generally, it is evident that the technique is applicable on domains where there are multiple choices of actions. Even the learning time might be manageable as well. Notice that the agent achieves 700 successes at episode 60,000 which is not too much more that the 9,000 episodes which it took to learn a similar length T-maze.

7.6 Summary and Reflections

The above experiments bring new light on the RL/LSTM technique and they show several important aspects which are relevant in theoretical but especially practical settings. I will summarize these below.

- Advantage learning has a significant positive effect on the system, especially compared to Q-learning. Very small values, $\kappa \in]0, 0.2[$, are significantly better than the larger, $\kappa \in]0.2, 1]$, indicating that it is important for the agent to have a clear distinguishing between Advantage values.
- Conversely, eligability traces has a significantly adverse effect on this problem. The reason for this is uncertain but perhaps the traces creates a more "blurred" picture of the rewards making it harder to distinguish the temporal relation between road sign and goal.
- Bakker's Directed Exploration technique is a powerful tool and far better than undirected techniques. Undirected techniques, however, are capable of learning the task but with very poor performance.
- With too few memory blocks the system is either unable to learn the problem (0 memory blocks in our case) or has difficulty. It seems as if the problem needs at least two blocks for good performance on this problem. With just one, it has to "cram" the complex information into one block which is harder.
- Too many memory blocks make little difference to the system. They neither help nor hurt the performance.
- Initial gate bias to the memory blocks is not that important as long as enough memory blocks are present. If this is not the case the initial values has a significant effect. This is consistent with the notion that too few memory blocks need to encode more complex information making them more susceptible to a bad extrema in parameter space.
- Bakker's reported maximal corridor length of 70 is not the limit for this kind of agent. At least a corridor length of 144 is solvable which proves that even in an RL setting the agent can breach long time lags. Also, the convergence time do not seem to become too bad even though the episode length grows.
- The agent seems to learn in "bursts" which is usual for RL at least in MDP settings. It is evident that this feature of RL carries to POMDP settings as well.

- The system is able to learn more complicated relations with 7 escaping corridors (one is the winning) in the junction. It does not learn this task completely but shows promising results in this direction.

Chapter 8

Experiments with Forget gates

So far we have mainly been concerned with the T-maze problem and the setup used by Bakker. As was evident, the forget gate was never of much use and the main point of this chapter is to bring it into action.

In this chapter I extend the RL/LSTM structure as Bakker introduced it, [Bakker, 04], with the forget gates introduced by Gers, [Gers,01]. This chapter is the main contribution of this thesis as it shows new and exciting possibilities with the RL/LSTM construction.

I introduce the approach with a description of Gers tests on the Embedded Reber Grammar (ERG). Next, I make a continuous version of our T-maze problem and show some experimental results with this construction.

I augment the T-maze test problem so it becomes of continuous nature and use it for further experimental studies. I show that it works and that it even is rather stable with regard to the success rate and convergence time as mentioned in the last chapter.

8.1 RL/LSTM and the role of Forgetting

The concept of forgetting is the same as resetting the system. As I mentioned in the preface automatic resetting of the system is a goal of machine learning as it removes this responsibility from the designer. Recall that we discussed "how" to reset and "when". Both of these are in RL/LSTM the responsibility of the forget gate.

I remind of the role of the forget gate in figure 8.1. Similar to the other two gates the forget gate learns which situations require it to become active and its task is to reset the cell state. If this is not done the cell state will cycle forever and make the cell unable to adapt to a new environment.

Seemingly, this could be a mere matter of wasting resources: Once the agent has moved past the point where the contents of the memory cell was useful, it ought to close the cell down forever (with the outgate) and from then on it will only be an inactive and superfluous part of the network. As

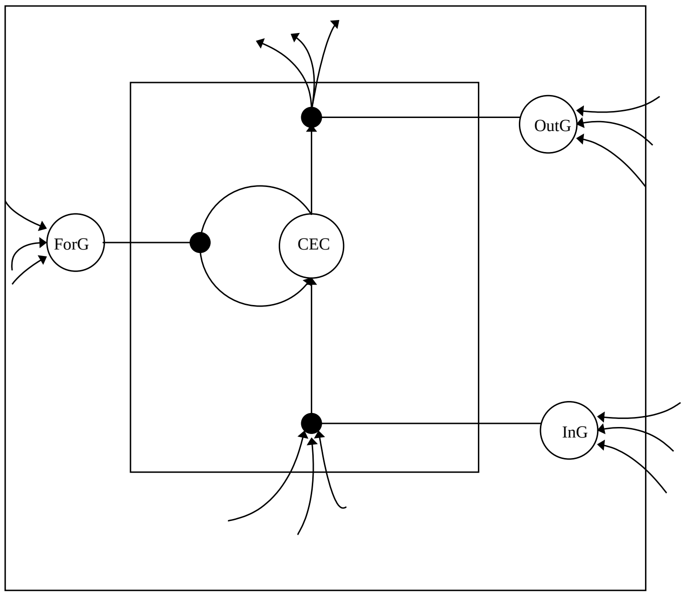

Figure 8.1: The role of the forget gate in the system is to reset the internal state of the cell at the right time.

shown earlier, a significant amount off superfluous memory blocks can be added without this in itself having an impact on performance.

The filled memory block is much more important than this, though. As we have made a significant effort to teach the memory block how to behave (ie. finding a useful set of edge weights), it is disasterous if we are to throw this away just because the cell state is invalid. At all costs we should seek to preserve a properly trained memory block.

These considerations let Gers to propose the forget gate in his dissertation [Gers,01]. His purpose was to allow a memory block to solve series of similar tasks without the necessity of a human or external actor which could reset internal states at appropriate moments.

8.1.1 Gers' Experiment

To get a perspective on the technique I will describe on of Gers' experiments which is a classical supervised learning problem for RNN.

Embedded Reber Grammar

Consider figure 8.2.

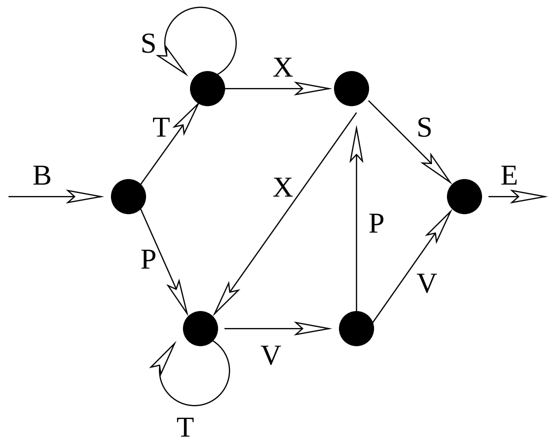

Figure 8.2: The Reber Grammar problem. A word is constructed from the letters on the edges. If there is a choice of the next edge, then each is chosen with 0.5 probability.

The task is to predict the next letter in a small language consisting of the letters 'B','T','S','X', 'V', 'P' and 'E'. Each "word" in the language is constructed by moving around the vertices in figure 8.2 and appending the visited letters to the word. When there are two possible vertices to visit the choice is determined randomly with 0.5 probability. In this way the word "BTSSXXTVVE" or "BPVVE" can be constructed but "BPVSPSE" cannot. This is called the **Reber Grammar (ERG)**.

Now consider the **Embedded Reber Grammar (ERG)** as shown on figure 8.3 which consist of a larger system with two standard Reber Grammar blocks included.

The task of the system is to predict the next letter when traversing a ERG-word. When the next letter is certain this should be predicted and when there is a choice the two choices should be predicted with equal weight.

The problem is the second-to-last letter, either 'T' or 'P'. These are certain but can only be predicted correctly if the second letter is remembered even though many similar letters (both 'T's and 'P's) may be present in the word.

Notice that the the smallest word is 9 letters long but these can be arbitrarily long with positive probability. The sequence of letters can be considered a Markov chain and Gers states that the expected string is 11.5 letters long but that the longest expected string is greater than 50 letters.

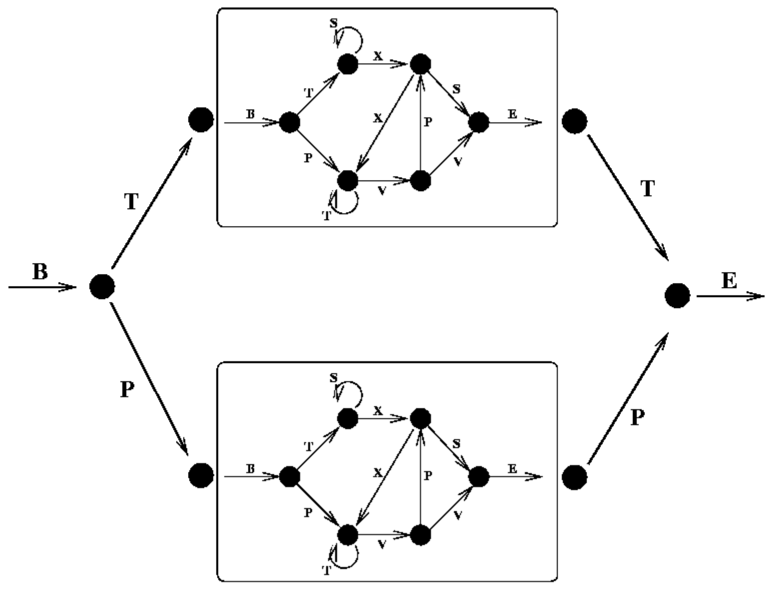

Figure 8.3: The Embedded Reber Grammar. After 'B' There is a choice between 'T' and 'P' followed by a Reber Grammar word. The task is to remember the 'T' or 'P' in order to predict the last two letters of the word ('TE' or 'PE') correctly.

This problem is a temporal problem and Gers shows [Gers,01] that LSTM solves this easily, fast and perfectly whereas most other recurrent algorithms (RTRL, Elman networks and others) fail completely or nearly so.

Continuous ERG

Gers makes the task continuous by constructing independent words and appending them to each other. This is equivalent to forming a connection in figure 8.3 from the last letter, 'E', and back to the first, 'B'.

In this new task there is no clear demarkation of the end of one word and the beginning of the next. In order to predict correctly the agent still have to remember the second letter in the word, but once it enters the next word the former information is *no longer valid* and needs to be changed according to the new word.

In Gers experiments the extended LSTM-network (with forget gates) massively outperformed conventional LSTM which, as expected, did not cope with this task very well. LSTM with "external resets" after each word performed best on the task. Interestingly, though, the extended LSTM performed almost as well (Traditional LSTM had 0 succeses. "Externally reset"-LSTM had 74% succeses and LSTM with forget gates had 62% succes).

Gers experiment will not play a further role in this thesis. As a curiosity, I reproduce his network design in figure 8.4 since it is rather different from the one I use.

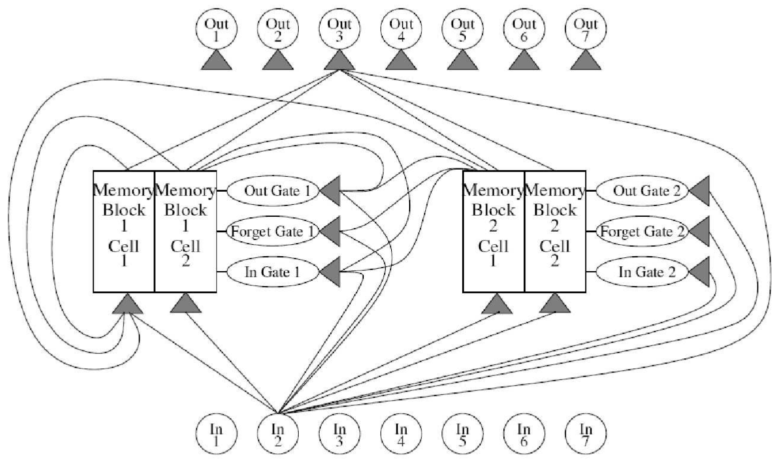

Figure 8.4: The network design used by Gers to solve the continuous ERG-problem. Notice the two cells per memory block. From [Gers,01].

8.2 Continuous T-maze

The main point of the extended LSTM network is that it enables the agent to solve many similar problems without external input as to where one problem stops and another starts. I will now turn to its use in our task.

I introduce the **continuous T-maze**. It is similar to the regular T-maze but every time the agent escapes the maze it is teleported back to the beginning of the T-maze. Upon arriving at the beginning, the maze is changed so that a new goal is determined randomly.

I let the agent move for as long as it can or until it makes a wrong turn at the junction. In that case the episode is declared "lost" the agent is reset and starts a new episode. If indeed the agent successfuly makes the correct action in 10 consecutive junctions the episode is a success and is terminated. 10 is an arbitrary choice and other choices will be examined below. Henceforth, I define the number of times the agent has to complete the maze as the number of **reruns**. I also refer to the individual mazes in the episode, where each rerun is made, as a **sub-maze**.

The maze is depicted in figure 8.5 in two representations. The one to the right is the one described above, the one two the left is a perhaps more intuitive "unfolding" of the maze.

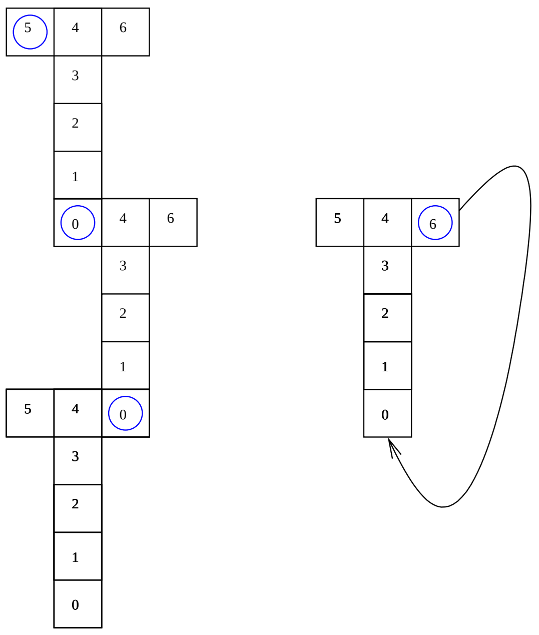

Figure 8.5: The continuous version of our T-maze. When making the correct choice in the junction a "new" T-maze is formed with a new random escaping corridor. If the agent makes the wrong choice at the junction the episode is terminated. To the left is the "unfolded" maze corresponding to 3 reruns and to the right is the original T-maze with a "teleporting" facility.

8.2.1 Different Lengths of the Corridor

I first test on varying corridor length. The reason for this is to explore the effect of longer time lags between the points of reset. The focus are the same as the one in the previous chapter: success rate with random initialization and convergence time. The number of reruns is 10.

Results

Due to testing time I terminated the test after a corridor length of 14 had been processed.

The success rate is depicted in figure 8.6 and the convergence time in figure 8.7.

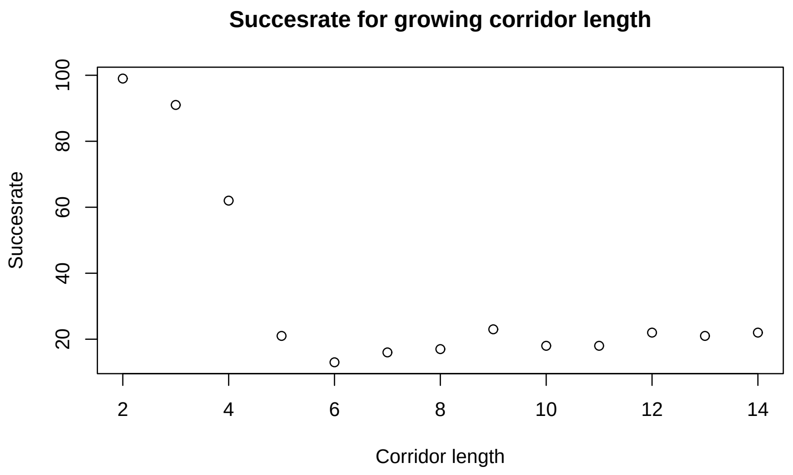

Figure 8.6: Success rate for the continuous T-maze with varying corridor length. At each maze the agent has to make 10 reruns in order to achieve succes.

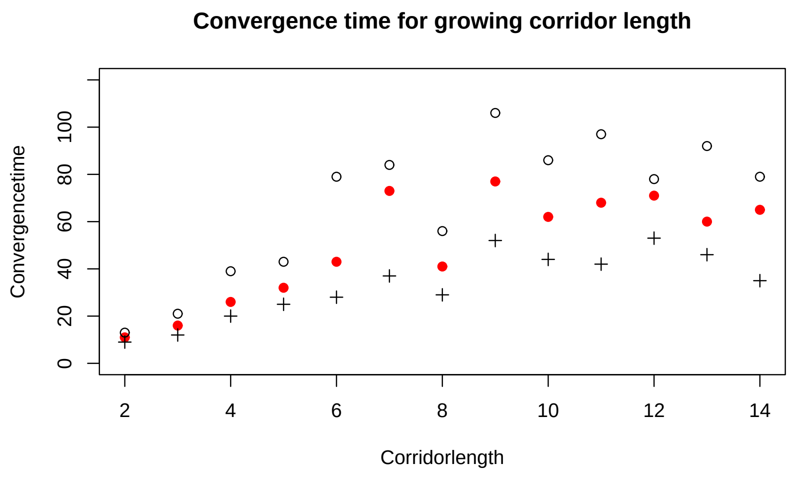

Figure 8.7: Convergence time for the continuous T-maze. The circles denote median convergence time, the triangles the 3rd quantile and the crosses the 1st quantile. Number of episodes is in thousands.

It is obvious that the problem is "more difficult" than the similar non-continuous test in the previous chapter. The success rate is lower and the number of episodes needed is much longer.

This is indeed no surprise as the task has become much more complicated. Not only is it necessary for the agent to learn a relation between road sign and junction but then it needs to learn a specific relation that is best solved by resetting the internal connection. First, it has to learn the task and to remember the road sign, then it has to learn how and especially when to forget.

Notice also that the 95% success rate perhaps is becoming too strict. If the agent has to choose correctly in 10 consecutive junctions it will at least have 10 situations with a 5% chance of not choosing the optimal action. Strictly speaking this will lead to a 60 % chance ($0.95^{10} = 59.8$) of "surviving" to the goal without one deviate action at each junction. Still, the suboptimal action do generally not lead to a lost episode and I have seen examples of perfect success rate even with several reruns.

8.2.2 Different Number of Reruns

In this section I test with a fixed corridor length of 4 and with a growing number of reruns. This is interesting as the agent will always be reset after the last rerun. With more than one rerun, however, the agent will have to learn to do this for it self. With a significant number of reruns we might see if the agent is able/unable to reset itself approapriately during the episode. I terminated the test after 25 reruns.

Success Rate

The success rate is depicted in figure 8.8. We see an initial loss of function-

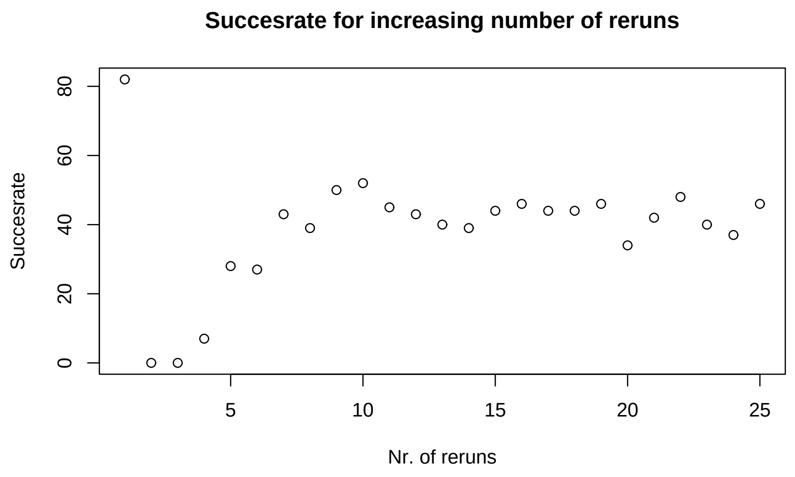

Figure 8.8: Success rate for test with corridor length of 4 and growing number of reruns.

ality with 2 or 3 reruns and then steadily improving to some level with 4 or more reruns. The reason is probably that it has more exposure to the environment when it is possible to make many successful reruns in one episode.

With just 2 reruns the agent has very limited data at its disposal to realize where to reset. What is not evident in the figure is that the agent generally learned the correct action at the first junction. It is therefore able to utilize the rest of the system (similar to the one used in the previous chapter) but unable to tune the forget gates correctly.

Convergence Time

The median convergence time is depicted in figure 8.9. The convergence times seem fairly stable and not too influenced by the number of reruns. Recall, however, that a successful episode with 10 reruns is 5 sub-mazes longer than one with 5 reruns. This doubles the amount of data for updating the system pr. episode.

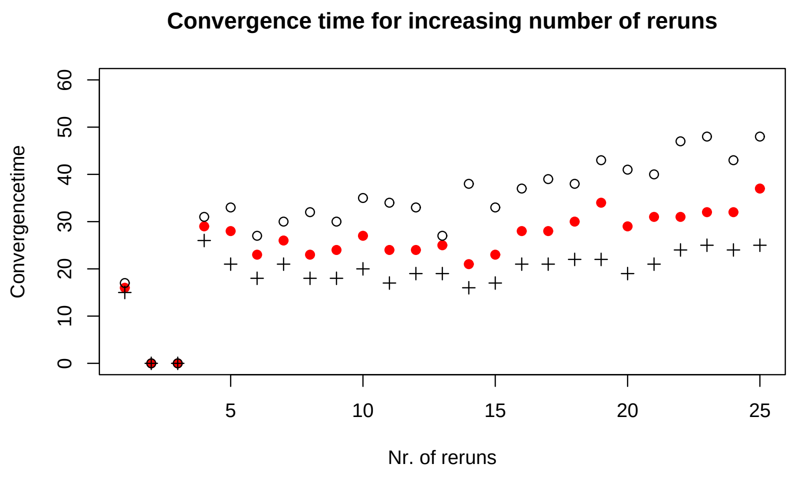

Figure 8.9: Convergence time for test with corridor length for 4 and 1 to 25 reruns. The filled red circles is the median convergence time and the crosses and open black circles are the 1st and 3rd quantile respectively

8.2.3 Removing intermediate Rewards

The agent in the previous experiment was actually helped somewhat by the reward function in the system. As the agent receives a bonus of 4.0 when winning the maze it will have an implicit notification of the termination of a task. As the number of reruns grows this will be evident many times during one episode.

This is not quite what we wish as the designer is then required to notice and mark the endings of tasks with a special reward in order for the system to achieve this kind of performance.

Below I repeat the experiment but this time without the reward when winning a maze during the episode but instead with a reward for completing all reruns correctly. The agent now receives a zero reward for winning a single maze which is still better than loosing (-0.1) but no different than moving south.

Results

The test was terminated after 14 reruns (only 10 is displayed), as every test run with more than 5 reruns was unsuccessful.

The success rate is depicted in figure 8.10 and the convergence rate in figure 8.11 The success rate drops rapidly with more reruns and convergence time increases similarly. This is understandable as the task has become harder to solve.

Notice that the problem is no longer about winning each sub-maze in the episode but rather to avoid *loosing* a sub-maze. In the previous experiment the agent has an easy way of realizing a good action to make at the junction

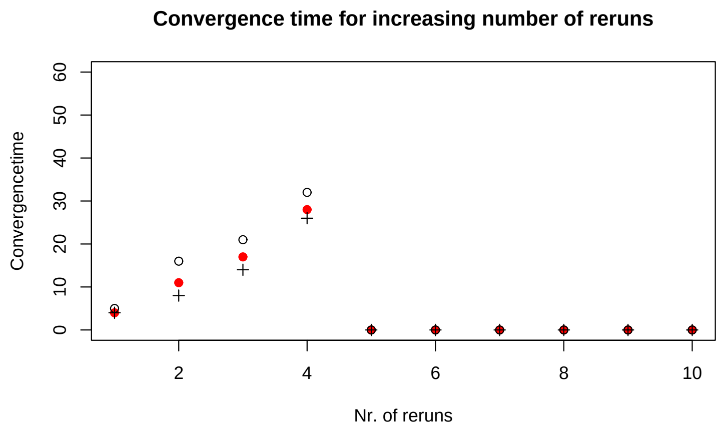

Figure 8.10: Success rate for test with corridor length of 4 and growing number of reruns. This time without rewards when taking the winning action at each junction.

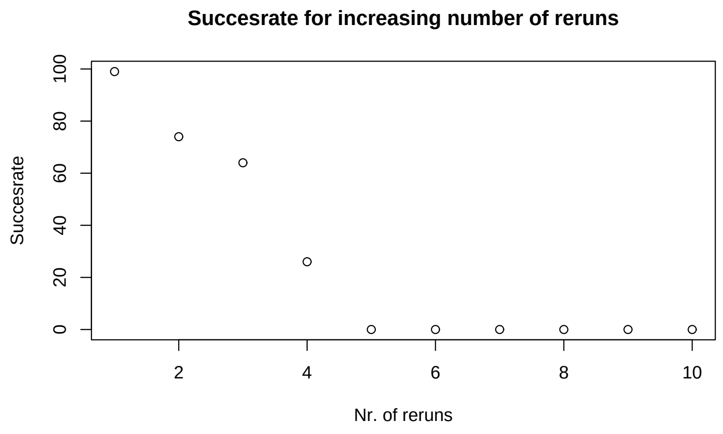

Figure 8.11: Convergence times for test with corridor length of 4, growing number of reruns and no rewards when making winning actions. The filled red circles is the median convergence time and the crosses and open black circles are the 1st and 3rd quantile respectively.

when this entailed a large reward.

Now, at the junction, the agent has to differentiate between the north action and loosing which both has -0.1 punishment and the south and winning action which both has zero reward. When a final reward is “close” (few reruns) this will help in realizing the correct action but it is not surprising that this help is too limited once the number of reruns grows too much.

Surprisingly, the agent is now able to learn with just 2 reruns as opposed to the last experiment. This is an interesting result and might stem from the fact that the agent in the previous test received a reward for *reaching a subgoal*. Such subgoals are dangerous (see the classical reading [Sutton & Barto,98]) since the agent might be more focused on achieving the immediate subgoal and disregard the global goal. Even if the global goal over time would be the larger.

8.3 Network analysis

It is useful to study the activation of the trained LSTM-network during an episode in greater details. The weights also change during an episode but as this is only a small change I consider them fixed.

In order to keep the presentation at a manageable level I do not present neither edge values nor the activation of all neurons during an episode. Rather, I focus on the most interesting ones: the output neurons and the memory blocks.

The test is a continuous T-maze with corridor length of 5 and 5 reruns. I use four memory blocks with one cell each and otherwise similar parameters as the rest of this chapter. I train an agent to perfect success rate and then examine the activation during one episode.

I compared the activation values below with other executions on the same problem and the agent seems consistent in its particular use of the network (ie. how different cells are used and when). With other parameters, however, (especially fewer memory blocks, different gate bias, etc.) other systems, where the memory blocks are used in a very different manner, will emerge. This coverage do still give a rough idea of the inner workings of the technique.

First, a note on the general format of the figures in this section. Refer to figure 8.12.

The x-axis is the time parameter and is divided into time steps. Even though it is not explicitly shown, the agent moves through the maze as fast as possible. That is through states 0,1,2,3 and 4 in that order in each sub-maze except the 3'rd and 5'th where it makes a single (non-optimal Advantage) action in the 0'th state and moves into a wall.

The maze is escaped and won in the 27'th time step as the agent chooses correctly in the 26th time step (only the first 26 is therefore shown). As the agent makes 5 reruns of the T-maze each rerun is noted on the figure with its corresponding winning corridor (East or West). Also, I separate each rerun with a vertical dotted line which is also drawn through those time steps where the agent is at the junction. I use this type of figure when describing the cells and gates as well.

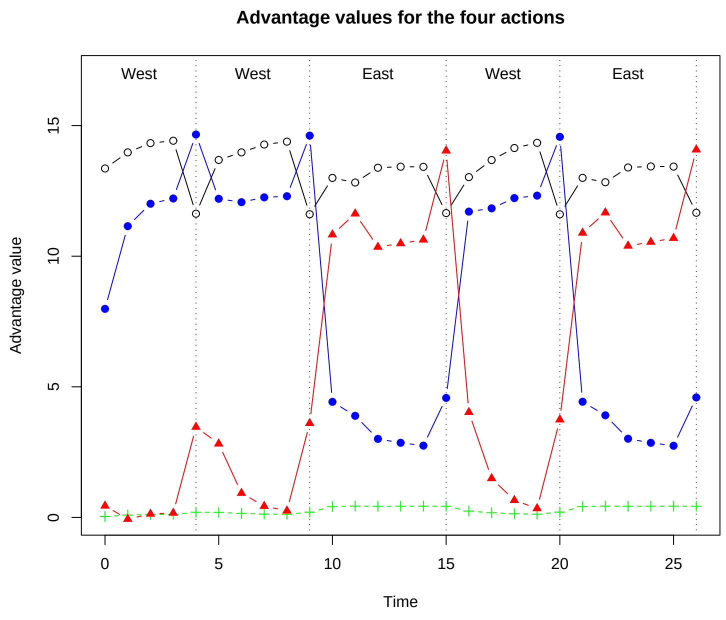

Figure 8.12: The activation of the output functions during an episode. The vertical dotted lines denote the arrival at the junction state. As the agent escapes the sub-maze after each junction state the dotted lines also separate reruns. The winning corridor for each rerun is noted at the top. The black open circles is the Advantage value of the North-action. The blue filled circles correspond to the West-action and the red filled triangles to the East-action. The green crosses denote the Advantage values of the South-action.

8.3.1 Advantage values during an episode

As the network approximates the Advantage function, the four values are the Advantage values of each action. The Advantage value is the y-axis on figure 8.12.

Several expectable behaviors are evident. The North-action (black open circles) has the highest advantage in all states *except* the junction state. In the junction state the North-Advantage (short for "Advantage value for the North-action") is also high which is probably due to the smoothing property of the function approximation. Still, the North-advantage is *always* lower than the East- or West-advantage in the junction.

The Advantage value of these two actions has a more or less random behavior in the hallway but always lower than the North-advantage. When the winning corridor is to the East the East-Advantage is higher and vice versa if the goal is to the West.

The South-Advantage is always much lower than all other Advantage values. It is indeed rather constantly at around 0 where the other Advantage values fluctuate more.

These Advantage values are the final output of the system and are used by the agent when deciding which action to make in each state. As is evident, they provide a rather precise notion of the optimal action to make. Whenever the agent do not choose optimally it is because it decides to make exploring actions.

8.3.2 Activations of the memory block elements

Next, I describe what goes on in the memory block during the episode. In general, all the work is being done by memory block 1 and 4. 2 and 3 seems not to contain relevant information during the episode.

In the figures below I use this convention for the different memory blocks. As there is only one cell pr. block I do not distinguish between cells and memory blocks explicitly.

- Elements of block 1 are black, filled circles.
- Elements of block 2 are blue, open circles.
- Elements of block 3 are green crosses.
- Elements of block 4 are red, filled triangles.

Cells

In the last section I dealt with the output of the entire system and the next step is the output of the memory system see figure 8.13. Recall that the output of the system is the squashed cell state multiplied with the outgate. This places the cell activation in the interval $]-1,1[$.

$$y^c(t) = h(state_c(t)) * y^{oG}$$

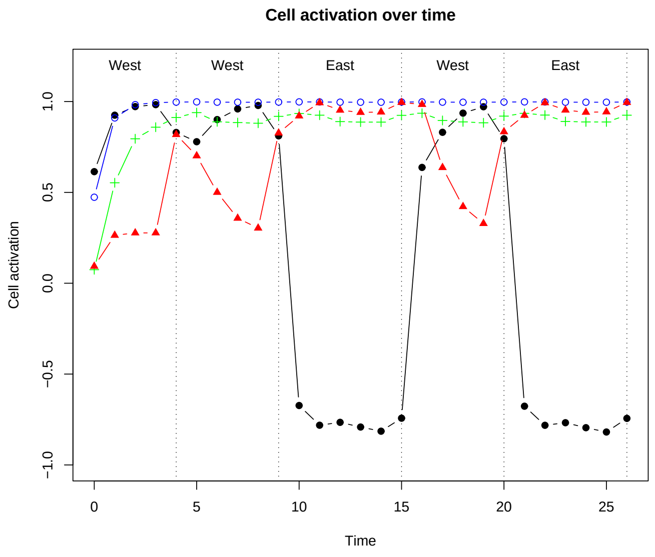

Figure 8.13: The activation escaping the memory cells during an episode. Notice how cell 1 is negative during the Eastern sub-mazes.

Block 2 and 3 (blue, green) is almost always sending positive information into the network. Block 4 (red) is behaving strangely and block 1 mirrors 2 and 3 except in Western sub-mazes where it becomes largely negative.

This behavior is seemingly strange. We might have expected that cell 1 and 4 in some way encode the two choices but this is not immediately obvious. A look on the internal cell states, see figure 8.14, confirms the "strange" behavior. The state of cell 2, 3 and 4 all diverges! With only cell 1 having an internal state which has an expected behavior. The explanation for this behavior is as follows.

First, the diverging cell states is less of a problem than it might seem. For once, they are all reset after 5 reruns and their output will also never grow above 1 as it is squashed.

Secondly, the cells *do* encode the information the agent needs but in a little different fashion than imagined. This agent has a *default* action whenever it reaches a junction and that is to move to the west. This information is primarily encoded in cell 4 but also somewhat in cell 1. *If* the goal is

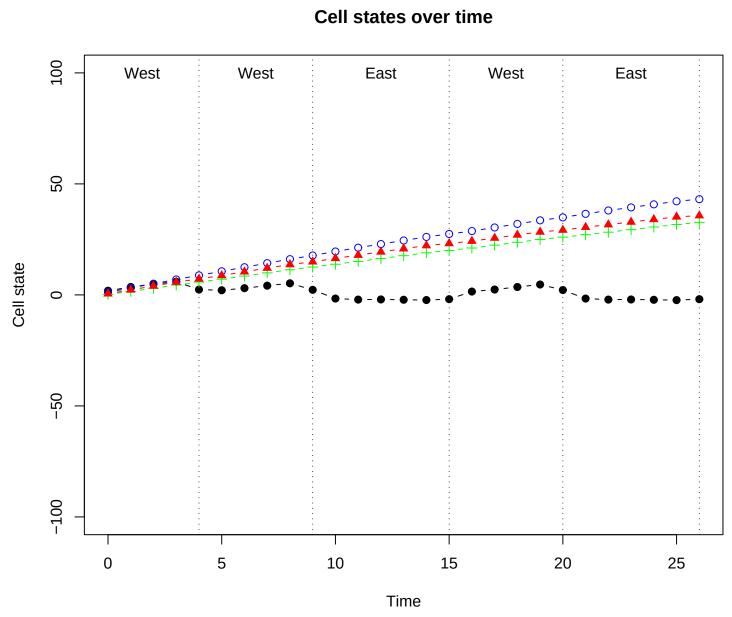

Figure 8.14: The internal cell state. Cell 2,3 and 4 has diverging cell states where as only cell 1 is somewhat well behaved.

to the east a special behavior is initialized and cell 1 will become highly negative even throughout the entire sub-maze.

When the agent is in the corridor the edge weights of the FNN system will make the North-advantage higher no matter what activation the memory cells do or do not emit. When in the junction the memory connections play a larger role and will have a tendency to favor the western choice *unless* cell 1 is highly negative in which case the eastern Advantage value will be the greater.

This behavior is similar to the one mentioned in the section on the 8-maze where one could imagine a system designed to behave as this:

> “if this specific assertion is true, then do this specific action.
> In all other cases behave as usual.”

Gates

With the behavior described in the last section the activations of the gates are easier to understand. The activation of the ingates are depicted in figure 8.15. The ingates are wide open throughout the episode. This is not unex-

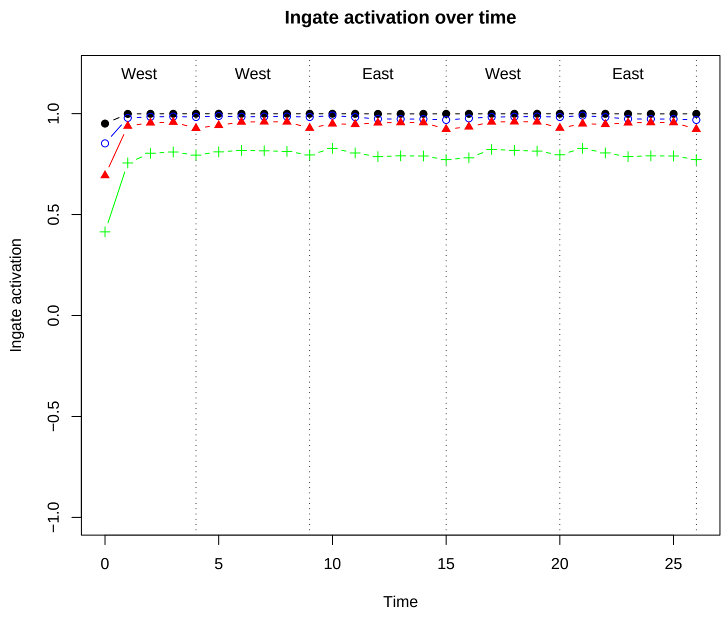

Figure 8.15: Activation of the ingates during an episode. They are not closed as might be expected.

pected when seeing the diverging cell states in figure 8.14. It is interesting that the ingates in this problem apparently are not needed. The system do control its internal state through the forget gate as explained below.

The activation of the outgates is depicted in figure 8.16. As with the ingates the outgates are strikingly open at all times except for block 4 which only opens at the *junction of the western sub-mazes*. It is also open (even throughout) eastern sub-mazes but in this case it is the responsibility of block 1 to produce the output needed for the correct action. In western sub-mazes block 1 plays no explicit role and it is the increased activation of cell 4 (the more open outgate) which is the crucial point in evaluating the correct Advantage value.

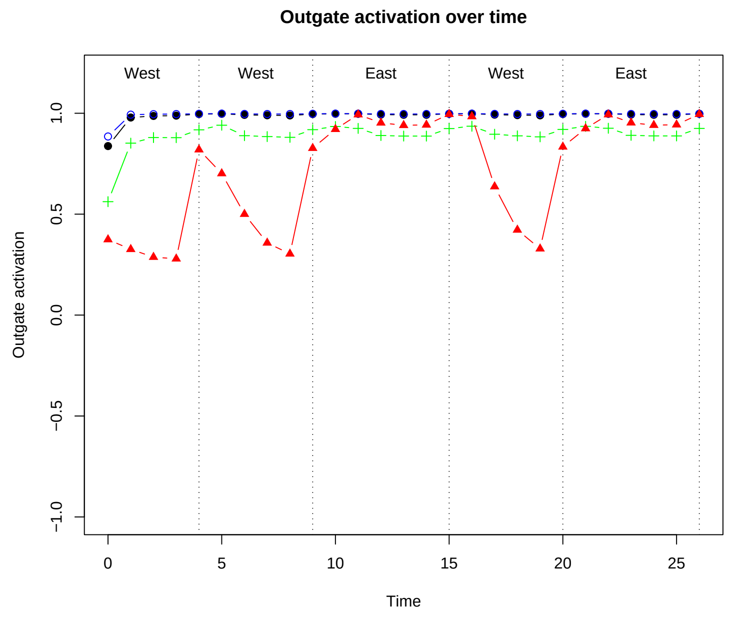

Figure 8.16: Activation of the outgates during an episode. As the ingates these are not closed except for the block 4 which is much more open in the junction than in the hallway.

Notice how the outgate is always open for block 2 and 3. I have said that useless memory blocks are shut down through their outgate. This indeed happens, but in this case the system always expect a 1-activation from these two cells (throughout the episode). So it has found it more convenient just to incorporate this, perhaps through a near-zero edge weight from cell 2 and 3 to all output neurons.

Lastly, we look at the activation of the forget gates in figure 8.17. These are all open (no forgetting) throughout the episode except for block 1 which is routinely reset at the end of each sub-maze.

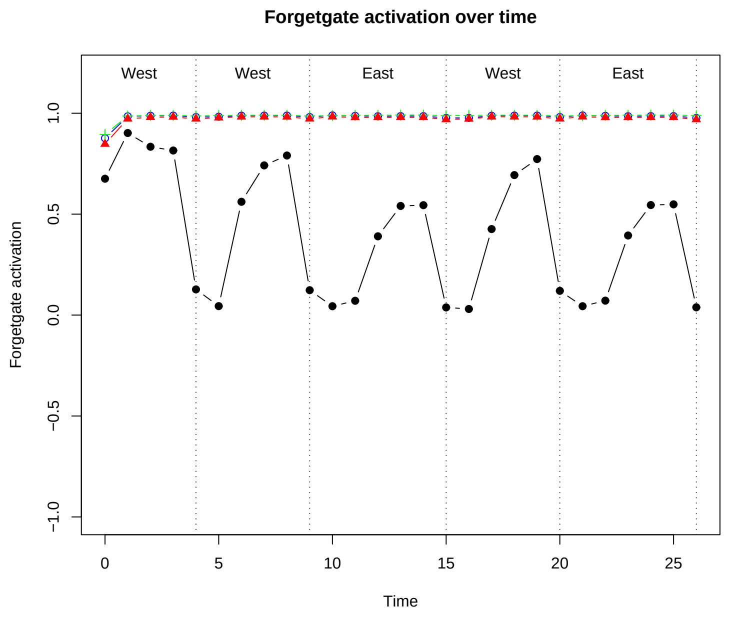

Figure 8.17: The forgetgate is unused for all blocks except block 1 which uses it to explicitly reset itself after each rerun.

This helps explaining the always-open ingate problem. Even though the ingate is always open the important information is systematically zeroed when needed. It seems as cell 1 (when looking at cell states figure 8.14) does not gather as much cell state as one might expect from the open ingate. It is obvious that cell 1's connections for net input is highly specialised and focused on the specific states concerning the eastern sub-mazes. In these sub-mazes cell 1's internal state is slightly negative (around -2.0) whereas it is slightly positive in western sub-mazes (growing to around 4 or 5). In order to change this sign a reset is necessary.

8.4 Summary and Reflections

In this chapter I have augmented RL/LSTM as devised by Bakker (see [Bakker, 04]) with forget gates as introduced by Gers (see [Gers,01]). I have tested this construction with a continuous version of the T-maze used in the last chapter.

The construction works and the system is able to:

1. Find the states where a reset of internal states is necessary.
2. Reset these states appropriately.

I test this with varying corridor length of the sub-mazes and show that not only the time dependency discussed in the last chapter but also the reset-functionality described in this chapter can be learned with time lags as long as 14 time steps (ie. a T-maze with corridor length of 14) for the individual sub-problems.

The estimated time to convergence, or learning, is growing but slowly enough to be useful in applications.

I test with varying number of reruns, or times the agent has to complete a sub-maze in an episode before being declared to have won. This is to show how effective the reset functionality is in the long run as the system is only reset on either winning or loosing the maze. Here, the system is able to complete 25 sub-mazes without the need for external resets, meaning that it can perform the intermediary resets of the internal state by itself. There is no indication that this is a limit, on the contrary the success rate with 25 reruns is around 50%. Also, the time to learning is very acceptable even with large number of reruns.

I also test with no specific reward at the point of termination of a sub-maze as this is a help for the system to learn where reset should be done. As expected, this proves much harder for the agent to learn but it succeeded with mazes requiring up to 4 reruns (4 internal resets). This seems to be a limit for this problem type but might be pushed with more extensive parameter tuning.

Note that with positive rewards so sparsely distributed it would probably prove to be a hard problem for any learning system, even human beings. Successful implementations might need to subdivide larger problems into smaller which could be solved by this technique.

Lastly, I show the inner workings of the memory of the agent during an entire episode with 5 reruns. The agent is very stable (1000 wins out of a 1000 episodes) and the inner workings clearly show how and when the different actions are favored or not.

Chapter 9

Concluding Remarks and Perspectives

I have in this thesis shown the use of RL/LSTM as Bakker introduced it in [Bakker, 04] and extended this technique with the ability to forget.

This creates an agent with a decision system with a feature very similar to short-term memory. Recall from the preface that I claim this because the agent:

1. Recognizes an important event and stores the relevant data from this event.

2. Keeps this information "safe" from being overwritten or altered for a while.

3. Recognizes another important event where it is necessary to "fetch" the information kept in the storage and use the stored information in decision making.

4. Overwrites the content of the storage with new information when necessary. Technically, the agent first deletes the content and stores new information but the net result is the same.

This agent is able to solve various POMDP problems of both non-continuous and continuous nature. The specifics are summarized in the next subsection on contributions.

9.1 Contributions

The technical contributions of this thesis is divided in two parts mirroring the coverage in chapter 6 and 7 respectively.

Explorations on the RL/LSTM structure

The following is a selection of elements from this thesis which I have not found in any literature. Each is important in its own right as it shows different aspects and possibilities with RL/LSTM. These features will likely prove beneficial in concrete applications.

1. I have shown that Advantage learning is a definite enhancement over Q-learning on the T-maze when using RL/LSTM. It ought to be a primary consideration when using both RL/LSTM or RL with function approximation.

2. I have shown that the use of Advantage(λ) has a substantial negative effect on the system when compared to standard Advantage learning. This surprising result should not discourage use of the method immediately, but suggests that care should be taken.

3. I have shown that directed exploration is very useful in this context, especially when compared to undirected exploration. While this is not surprising in theory, it shows that Bakker's Directed Exploration technique is focusing on many of the right parts of the state space.

4. I have shown that the number of memory blocks is a "benign" parameter: while too few do entail some difficulties, too many (even far too many) do not seem to impact system performance in any significant way. This seems to hold for the T-maze, but is likely to carry to a more general setting as it seems to be a feature of the LSTM system.

5. Bakker's reported maximal corridor length of 70 is not the limit for this kind of agent. At least a corridor length of 144 is solvable which proves that RL/LSTM can breach relatively long time lags. This is even with little impact on the convergence time.

6. The system is able to at least partly learn more complicated relations with 7 escaping corridors (one is the winning) in the junction.

Especially the part of Advantage(λ) is troublesome. Usually the introduction of eligability traces is a benefit for the system, especially with regard to convergence time. Even though the opposite is the case in the T-maze it is certainly too early to dismiss the technique.

The main question is why it fails to function. Is it the specific properties of the T-maze? or POMDPs in general? Does it have something to do with RL/LSTM? or perhaps the use of function approximation?

The question is important as there is some underlying aspect which malfunctions with eligability traces. Until this facet is uncovered and explored it will most likely resurface in other applications as well. For now, it will suffice to use Advantage(λ) a little cautiously and to bear in mind that it might not be beneficial in one's application.

Extended LSTM and RL

The use of extended LSTM (with forgetgates) is new to this field and experiments show that the construction is both robust and useful. Particular facets of this are the following results.

1. I show that the RL/LSTM structure extended with forget gates can solve a continuous version of the T-maze. It is able to discern the point in time where one task is completed (one maze) and another begins and also to reset itself appropriately at this point.

2. I show that this structure can handle rather long time lags in the individual sub-tasks (at least 14 time steps)

3. I show that the reset functionality is rather stable with excellent behavior even after 25 internal resets.

9.2 Further Research

Below I have gathered a few points which would be particularly interesting to explore further.

- **A Practical Implementation in a Robot**.
  A natural extension of the simulated experiments in this thesis is to use the technique in a real-world robot. As the goal of all AI is ultimately practical implementations this is a primary choice of research.

  The obvious obstacle is of course the need for many iterations until convergence. A promising approach, however, is to let the robot experience some of the environment and then "iterate" this sampled data internally. This procedure could be repeated until the robot performs acceptably. See [Bakker et al.,03] for a successful implementation of this.

- **A practical implementation in non-trivial simulated environments**.
  With games like Whist, Bridge or Hearts, it should be possible to struggle with interesting temporal problems which are still easy to understand and analyze. Here, there are less strict limits on iterations and learning time.

- **Explore Advantage(λ) learning and eligability traces in general in a POMDP setting.**
  The discouraging results with Advantage(λ) in this thesis is contrary to the results obtained in classical RL. An investigation of this is important as it is unclear what parameter in the system (the problem, RL/LSTM, function approximation, etc.) is working poorly with eligability traces.

- **Experiments with more cells pr. memory block** The computational or pattern matching power of more cells pr. block is an interesting facet to explore. While [Gers,01] shows interesting and promising results with a two-cell pr. memory block approach, it is still a open question what effect more cells pr. block has.

To emphasize the main point of the thesis I remind of the quote from the preface:

> Ask not what you can do for your robot;
> Ask what your robot can do for you.

In order for AI or machine learning to become a more attractive solution method, it is necessary to let the system do as much work as possible and limit the toll on the system designer.

There is nothing more easy than to nurse an agent throughout its life and feed it with all the right experiences at the right time and shield it from the wrong parts of the world. This is not an effective way to raise human beings, or any animal for that matter, and neither is it for an artificial agent.

RL is an approach to counter this by limiting the amount of teacher intervention which is so common in supervised learning. Classical RL has drawbacks, however, as it is limited to MDP-based problems.

The integration of LSTM with RL is a promising way of extending RL to the POMDP domain and I have in this thesis shown that the RL/LSTM construction is both versatile and robust.

When extending RL/LSTM with forgetgates it is possible to reduce the responsibilities of the designer even further as the system is capable of re-setting itself at the appropriate moments during execution. This enables the system to be more appropriate for long-term use, perhaps in environments which are not observable by the designer or other external entities.

This makes extended RL/LSTM an ideal platform for an agent to learn how to behave in a complicated environment and I look forward to see the aspects of this in the future.

Appendix A

LSTM Pseudocode

This is the LSTM activation and learning algorithm. It is adapted from gers [Gers,01] but with the removal of some superflous loopings and without peep-holes. Partials are denoted dS

LSTM Algorithm

```
1   reset: $state_c \leftarrow 0, \forall j \quad y^j \leftarrow 0, dS \leftarrow 0$
    **Forward Pass:**
2   $y^i \leftarrow input_i$
3   $\forall h \quad y^h = f(net_h); net_h = \sum_j w_{hj} y^j$ //hidden layer
4   **for** each memory block
5           **do** $net_{iG} = \sum_j w_{iGj} y^j, y^{iG} = f(net_{iG})$
6              $net_{iG} = \sum_j w_{iGj} y^j, y^{iG} = f(net_{iG})$
7              $net_{oG} = \sum_j w_{oGj} y^j, y^{oG} = f(net_{oG})$
8              $net_{fG} = \sum_j w_{fGj} y^j, y^{fG} = f(net_{fG})$
9              **for** each cell $\nu$ in block
10                 **do** $net_c^\nu = \sum_j w_{cj}^\nu y^j$
11                    $state_c^\nu = state_C^\nu y^{fG} + g(net_c^\nu) y^{iG}$
12  **for** each memory block
13          **do for** each cell $\nu$ in block
14                 **do** $y_c^\nu = h(net_c^\nu) y^{oG}$
15  $\forall o \quad y^o = net_o = \sum_j w_{oj} y^j$
16  **for** each memory block
17          **do for** each cell $\nu$ in block
18                 **do** $dS_{cj}^\nu = dS_{cj}^\nu y^{fG} + g'(net_c^\nu) y^{iG} y^j$
19                    $dS_{iGj}^\nu = dS_{iGj}^\nu y^{fG} + g(net_c^\nu) f'(net_{iG}) y^j$
20                    $dS_{fGj}^\nu = dS_{fGj}^\nu y^{fG} + state_c^\nu f'(net_{fG}) y^j$
    **Backward Pass:**
21  $e_k =$ injected error
22  $\forall o \quad \delta_o = f'(net_o) e_k$
23  $\forall h \quad \delta_h = f'(net_h) \sum_{j=o} w_{jh} \delta_j e_j$
24  **for** each memory block
25          **do** $\delta_{oG} = f'(net_{oG})(\sum_c state_c \sum_{j=o,h} w_{jc} \delta_j$
26             **for** each cell $\nu$ in block
27                 **do** $e_{state_c^\nu} = y^{oG}(\sum_j w_{jc\nu} \delta_j)$
28  $\Delta w_{ij} = \alpha \delta_i y^j$
29  **for** each memory block
30          **do** $\Delta w_{oGj} = \alpha \delta_{oG} y^j;$
31             $\Delta w_{iGj} = \alpha \sum_c e_{state_c} dS_{iGj}^j$
32             $\Delta w_{fGj} = \alpha \sum_c e_{state_c} dS_{fGj}^j$
33             **for** each cell $\nu$ in block
34                 **do** $\Delta w_{c^\nu j} = \alpha e_{state_c^\nu} dS_{cj}^j$
```

Appendix B

Reinforcement Learning Fundamentals

B.1 Basic Definitions

Reinforcement Learning (RL) is a machine learning methodology which is inspired by the psychological learning process of operant conditioning. The thorough, classical textbook on this subject is [Sutton & Barto,98] which covers the essentials in depth. Hansen's and my work [Nielsen & Hansen, 05] gives a broad coverage of RL in large state spaces with a special focus on the use of function approximators. This appendix can not replace the contents of especially Sutton and Barto's work but should refresh some of the main concepts.

The main idea is to have an **agent** which interacts in an **environment**. The agent passes through a series of environmental **states** and in each state it chooses an **action**. When performing this action the agent interacts with the environment and this results in a new state.

This setup is very general. A state could be sensor input for a robot, the positioning of pieces in a board game or the current situation of the real-estate market. An action might be the activation of motors, moving a piece or selling a house.

Upon performing an action and arriving at a new state the agent receives a reward which describes the "goodness" of arriving at this particular state. Rewards in the immediate future is usually perceivable but the goal for agent is to maximize rewards over time. It might therefore be necessary to walk through several low-reward states in order to achieve a high reward.

RL consists of the following:

- A **state space** S which in this thesis is finite but in general can be infinite as well. When finite we may talk of explicit and different states. When infinite we consider states to "continuously" flow into each other like for instance distance sensor readings for a robot.

- A finite **action space** A which also may be infinite. When finite we distinguish explicitly between actions like “move forward” or “turn left”. In infinite action spaces we will rather consider concepts like “injecting more fuel”, “apply more/less force” etc.
- A **state shift function**, $f : S \times A \to S$. This function is the underlying system dynamics and is the interaction between the agent and the environment upon performing an action. This function might be stochastic.
- A **reward function**, $R : S \to \mathbb{R}$. This function produces the reward of any state and might be stochastic as well.
- A **policy**, $\pi : S \to A$, which decides what action to take at state s. Essentially, this is the “brain” of the agent as this is the place where decisions are made. To improve the decision making (to learn) is in RL terminology also known as improving the policy. This function is most often stochastic in non-trivial applications.

In this way the agent will “produce” a series of states and actions where each state, s, leads to a choice of action, $a = \pi(s)$, and this again leads to a new state, $s' = f(s, a)$, with a reward, $r' = R(s')$.

If the series of states, the **path** or **history**, is finite it is called an **episodic problem** and the entire path is called an **episode**. If the path has infinitely many states it is called a **continuous task**.

An agent will seek to maximize the reward over time. In order to reason about this it is beneficial to use the **return** or **discounted return**

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{T} \gamma^k r_{k+t+1} \tag{B.1}$$

which is the return which the agent will receive from state s_t. $T = \infty$ is allowed. The **discounting factor** $\gamma \in [0, 1]$ is an adjustable parameter which assures that the sum will be finite (as the rewards can be considered bounded). For episodic problems it is acceptable to use $\gamma = 1$ but sometimes it is still useful to use $\gamma < 1$.

Notice that the return involves the actual rewards the agent gets. These are not alterable. In general, however, we mostly talk of the **expected return**, meaning the return we are likely to receive. The expected return for different actions and states is what guides the agent in its decision making process. A classical RL problem is depicted in figure B.1.

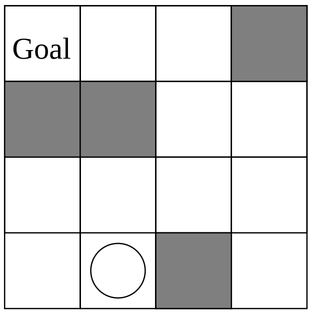

Figure B.1: A classic example of RL: Find the path to the goal. The agent receives a reward of 1 when the goal is reached and 0 otherwise.

B.2 Value functions and the Bellman equation

For an agent with a policy π and a given state, s, we define the **state value function**,$V^\pi : S \mapsto \mathbb{R}$, as

$$V^\pi(s) = E^\pi\left\{R_t|s_t = s\right\} = E^\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{k+t+1}|s_t = s\right\}\ , \tag{B.2}$$

This is the expected return the agent receives from being in state s and then follow policy π. The E denotes the mean value as π, the reward function and the state transition function all might be stochastic. Notice that for different policies we get different value functions on the same state space.

Another value function is the **action value function**, $Q^\pi : S \times A \mapsto \mathbb{R}$, which is defined as

$$Q^\pi(s,a) = E^\pi\left\{R_t|s_t = s, a_t = a\right\} = E^\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{k+t+1}|s_t = s, a_t = a\right\} \tag{B.3}$$

The difference is that the action value function incorporates the action taken as a parameter.

Value functions are used to estimate the value of different actions in a state. The action with the highest estimated value can then be chosen.

The **Bellmann equation** explains a special recursive relationship between the value function in one state,s, and the following state, s'. The Bellmann equation for the state value function is

$$\begin{aligned}
V^\pi(s) &= E_\pi\left\{R_t|s_t = s,\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma\sum_{k=1}^{\infty}\gamma^k r_{k+t+1}|s_t = s\right\} \\
&= \sum_{a\in A}\pi(s,a)\sum_{s'\in S}\mathcal{P}^a_{ss'}\Big[\mathcal{R}^a_{ss'} + \gamma E_\pi\{\sum_{k=0}^{\infty}\gamma^k r_{k+t+2}|s_{t+1} = s'\}\Big] \\
&= \sum_{a\in A}\pi(s,a)\sum_{s'\in S}\mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \qquad \text{(B.4)}
\end{aligned}$$

There exists a unique state value function which respect

$$\forall s \in S \quad V^*(s) \quad = \quad \max_{a \in A} \sum_{s' \in S} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{B.5}$$

This function is called **the optimal state value function,** V^*. From this an **optimal policy** can be derived simply by constructing a policy which for each state, s, chooses the action which leads to the best possible next state, s'. The goal is generally to find the optimal policy which is usually done by trying to find the optimal value function.

In order to find an optimal policy the common approach is **policy iteration**.

First, we initialize the agent with some policy, π, which need neither be optimal nor very good at all. Next, we evaluate the state value function for this policy by using **policy evaluation**

$$V^\pi_{k+1}(s) \quad = \max_{a \in A} \textstyle\sum_{s' \in S} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi_k(s')] \tag{B.6}$$

Where we have turned Bellmans optimality equation into an update in which the $k+1$ iteration builds on the value of the k'th. [Sutton & Barto,98] shows that this process will converge to the state value function for π even for arbitrary starting values for $V(s)$.

The next step is to perform **policy improvement** on the policy π, which amounts to finding a state, s, where our policy π chose an action which is not optimal according to the state value function. We then change π to π' which is similar to π except that in state s it choose the better action than π did. Note that after policy improvement the old value function is useless as it was derived from π.

By iterating between policy evaluation and improvement it can be shown [Sutton & Barto,98] that the policy and value function will converge to the optimal ones.

Appendix C

Neural Network Fundamentals

Here, I will briefly describe the Neural Networks (NN), their architecture, activation and training. Neural networks came to life from a biological inspiration: The neurons and synaptic connections in the human or animal brain.

C.1 Network design

Consider an acyclic, directed graph which is known as the **network**. In the network all vertices are called **neurons** to mirror the biological equivalent. Similarly, edges could be called synapses or dendrite/axons but they are usually just referred to as edges. This is depicted on figure C.1.

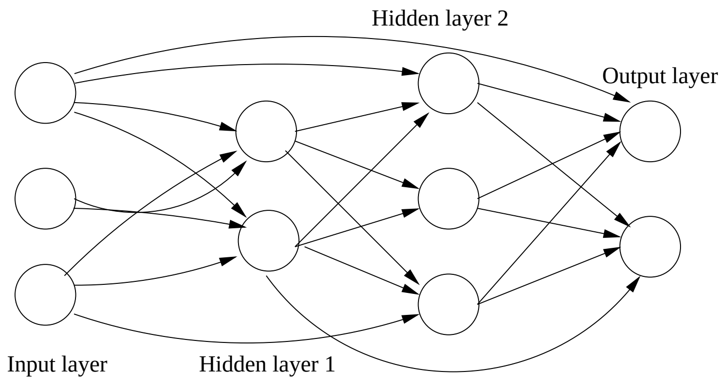

Figure C.1: The general neural network is an acyclic graph where the vertices are called neurons. These can be grouped into layers with the input and output layer accessible from the outside and with 0 or more hidden layers.

The neurons are grouped in **layers**. The **input layer** receives input from outside the system and the **output layer** sends information out of the network. These "external entities" could be sensors, motors, other neural nets or simply other parts of the program.

The important part of an edge is its **edge weight**. This is a real value and is termed w_{ij} where i denotes the *destination* neuron and j the *source* neuron. All edge weights are considered entries in a $N \times N$-matrix where N is the number of neurons. If two neurons have no connection the corresponding entry is zero. Notice that we neither use the upper triangle of this matrix nor the diagonal since the graph is acyclic and the use of these would create cycles.

The Feed-Forward Neural Network architecture (FNN) is depicted in figure C.2.

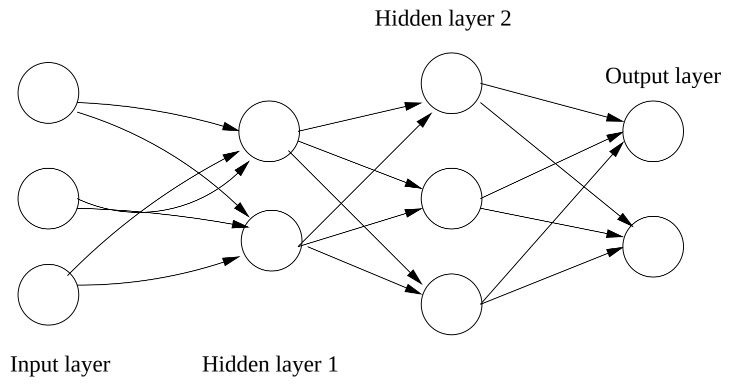

Figure C.2: The Feed-Forward Neural Network. Neurons in a layer is only connected to neurons in the next layer..

C.2 Activation

An neuron, indexed by j, has a **net input**, net_j, and an **activation**, y^j. This notation is fairly common but many others are used as well. The y^j has nothing to do with exponentials but it is sometimes convenient to "leave some space" for subscripts.

The net input is in general a real value but is sometimes restricted to subsets. The net input for the input neurons is set externally and will in this thesis be either "1" or "0". For all other neurons the input is

$$net_j = \sum_i w_{ij} y^i$$

Where i ranges over all neurons. Notice that even if the connections between neuron j and i does not exist, the formula still makes sense as we would implicitly have $w_{ij} = 0$.

The activation of any neuron,j, is $y^j = f(net_j)$ where the choice of $f : \mathbb{R} \to \mathbb{R}$ is rather open. Usual choices are

$$f(x) = 1_{x \geq c}(x) \qquad \text{Threshold function with threshold c.} \qquad (C.1)$$

$$f(x) = \frac{1}{1 + e^{-x}} \qquad \text{Standard sigmoid function} \tag{C.2}$$

$$f(x) = x \qquad \text{Indentityfunction} \tag{C.3}$$

Notice that the first two is never larger than 1 or smaller than 0. **Activating the network** is the process of assigning an activation to all input neurons and then computing net_j and y^j for all neurons. The "result" of the computation is the activation of all output neurons. While the activation order might in theory be arbitrary, it is common first to compute net_j and y^j for all neurons in the first hidden layer, then the second and so on.

C.3 Training

Training an NN amounts to changing the values of the edges. All other parameters of the network (activation functions, connections, even the input activation) are considered fixed.

The approach is usually to have a certain amount of training data, where each training **sample**, an encoding for the input layer, has an accompanying **label** which describes the expected activation of the output layer. If the network do not achieve this activation pattern among its output neurons, the network is considered to have made an **error**. Training seeks to reduce the cumulative error over all training samples rather than just a particular sample. A common approach to this, Backpropagation, is explained in the main text.

Bibliography

LSTM & Neural networks

[Gers,01] Felix A. Gers *Long Short-Term Memory in Recurrent Neural Networks.* PhD thesis. Department of Computer Science, Swiss Federal Institute of Technology, Lausanne, EPFL, Switzerland, 2001.

[Schmidhuber & Hochreiter, 95] Jürgen Schmidhuber and Sepp Hochreiter *Long Short-Term Memory.*Technical report FKI-207-95, 1995.

[McLelland, 86] . Rumelhart, Mclelland *Parallel Distributed Processing* volume I & II., 1986.

[Hochreiter, 91] S. Hochreiter *Untersuchungen zu dynamischen neuronalen Netzen.* Diploma thesis. Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[New AI] http://www.idsia.ch/ juergen/newai/newai.html

[Ziemke, 99] Tom Ziemke *Remembering how to behave: Recurrent neural networks for adaptive robot behavior.* From "Recurrent Neural Networks: Design and Applications". CRC press, 1999.

[Mehrotra et al, 96] Kishan Mehrotra, Chilukuri K. Mohan og Sanjay Ranka: *Elements of Artificial Neural Networks.* The MIT press. London, England, 1996.

[Robinson & Fallside] Robinson, A.J. and Fallside F. *The utility driven dynamic error propagation network.* Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department. 1987.

[Williams & Zipser] Williams, R. J. and Zipser, D: *Gradient-based learning algorithms for recurrent networks and their computational complexity.* In Y. Chauvin and D.E.Rumelhart: "Back-propagations: Theory, Achitectures and Applications". Hillsdale, NJ: Erlbaum. 1995.

[Bakker et al.,03] B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber: *A Robot that Reinforcement-Learns to Identify and Memorize Important Previous Observations* (PDF). In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS2003, 2003.

Reinforcement Learning (and RL/LSTM)

[Nielsen & Hansen, 05] Kristian Lerche Nielsen and Uffe Gorm Hansen. *Reinforcement learning på store tilstandsrum. en gennemgang med praktisk anvendelse* (in danish) graduate student project, May 2005. ftp://ftp.diku.dk/diku/image/publications/README.html

[Bakker, 04] Pieter Bram Bakker *The State of Mind, Reinforcement Learning with Recurrent Neural Networks.* Phd thesis. Universiteit Leiden, 2004.

[Bakker, 02] Bram Bakker *Advantage(λ) learning.* IDSIA, Switzerland. 2002.

[Sutton & Barto,98] Richard S.Sutton og Andrew G.Barto: *Reinforcement Learning, an introduction.* MIT press. London, England ,1998.

[Harmon & Baird, 96] Harmon, M.E. & Baird, L.C. *Multi-player resitual advantage learning with general function approximation.* Technical report No WL-TR-1065. Wright-Petterson Air Force Base Ohio: Wright Laboratory. 1996.

[Harmon, 94] Harmon, M.E. *Reinforcement learning in continuous time: Advantage updating.* Proceedings of the internationl conference on neural networks. 1994.

[DIKU homepage] www.diku.dk

Psychology

[Body and Mind Homepage] http://www.ku.dk/Priority/Body_and_Mind/index.htm

[Gazzaniga, et al. 02] Gazzaniga, M.S., Ivry, R., & Mangun, G.R. *Cognitive Neuroscience: The Biology of the Mind.* W.W. Norton, 2002. 2nd Edition