Large Scale Reinforcement Learning
using \( Q\text{-SARSA}(\lambda) \) and
Cascading Neural Networks

M.Sc. Thesis

Steffen Nissen
<lukesky@diku.dk>

October 8, 2007
Abstract

This thesis explores how the novel model-free reinforcement learning algorithm Q-SARSA(λ) can be combined with the constructive neural network training algorithm Cascade 2, and how this combination can scale to the large problem of backgammon.

In order for reinforcement learning to scale to larger problem sizes, it needs to be combined with a function approximator such as an artificial neural network. Reinforcement learning has traditionally been combined with simple incremental neural network training algorithms, but more advanced training algorithms like Cascade 2 exists that have the potential of achieving much higher performance. All of these advanced training algorithms are, however, batch algorithms and since reinforcement learning is incremental this poses a challenge. As of now the potential of the advanced algorithms have not been fully exploited and the few combinational methods that have been tested have failed to produce a solution that can scale to larger problems.

The standard reinforcement learning algorithms used in combination with neural networks are $Q(\lambda)$ and SARSA(λ), which for this thesis have been combined to form the Q-SARSA(λ) algorithm. This algorithm has been combined with the Cascade 2 neural network training algorithm, which is especially interesting because it is a constructive algorithm that can grow a neural network by gradually adding neurons. For combining Cascade 2 and Q-SARSA(λ) two new methods have been developed: The NFQ-SARSA(λ) algorithm, which is an enhanced version of Neural Fitted $Q$ Iteration and the novel sliding window cache.

The sliding window cache and Cascade 2 are tested on the medium sized mountain car and cart pole problems and the large backgammon problem. The results from the test show that Q-SARSA(λ) performs better than $Q(\lambda)$ and SARSA(λ) and that the sliding window cache in combination with Cascade 2 and Q-SARSA(λ) performs significantly better than incrementally trained reinforcement learning. For the cart pole problem the algorithm performs especially well and learns a policy that can balance the pole for the complete 300 steps after only 300 episodes of learning, and its resulting neural network contains only one hidden neuron. This should be compared to 262 steps for the incremental algorithm after 10,000 episodes of learning. The sliding window cache scales well to the large backgammon problem and wins 78% of the games against a heuristic player, while incremental training only wins 73% of the games. The NFQ-SARSA(λ) algorithm also outperforms the incremental algorithm for the medium sized problems, but it is not able to scale to backgammon.

The sliding window cache in combination with Cascade 2 and Q-SARSA(λ) performs better than incrementally trained reinforcement learning for both medium sized and large problems and it is the first combination of advanced neural network training algorithms and reinforcement learning that can scale to larger problems.
Preface

This is a master thesis from the Department of Computer Science at the University of Copenhagen (DIKU). The work was conducted by Steffen Nissen and was finished in October 2007.

My interest in neural networks and artificial intelligence was sparked by a course in Soft Computing taught by Peter Johansen. Peter has been one of the main forces in establishing the field of artificial intelligence and autonomous robots at DIKU, and as my advisor for this thesis he has made sure that my thesis kept its focus, and used his extensive knowledge within many areas of computer science to put my thesis in a broader context. I will be one of the last students to write under the guidance of Peter, since he will be retiring soon. I wish him the best of luck with his retirement and hope that he will keep in contact with the university.

In November 2003 I released the first version of my open source neural network library: Fast Artificial Neural Network Library (FANN)\(^1\), and I completed the paper describing the implementation (Nissen, 2003). The FANN library is today widely used in the field of neural networks, and is downloaded approximately 3000 times a month. A motivation for implementing this library was that I wanted to use it in combination with reinforcement learning to create a learning Quake III game bot (Waveren, 2001) for my thesis. It was with this plan in mind that I started taking a closer look at reinforcement learning and more advanced neural network training algorithms, and I quickly realized the potential of combining reinforcement learning methods with some of the advanced neural network training algorithms; the Cascade-Correlation algorithm in particular seemed promising. As I got more and more involved with this subject, I realized that I found the reinforcement learning and neural network aspect more interesting than the computer game aspect, and this realization lead me to start the work on this thesis. Little did I know, that in Canada François Rivest and Doina Precup, was working on the exact same combination of reinforcement learning and Cascade-Correlation (Rivest and Precup, 2003).

The work on the thesis has been long and hard but also very rewarding with periods of intensive research, long hours of coding and debugging, periods of intensive writing and weeks of writers block. During the work on this thesis I have learned a lot about subjects that I previously knew very little about. I dove head first into the large sea of reinforcement learning algorithms, only to realize that it would take years to investigate all the corners of the subject, forcing me to focus mostly on model-free methods. I have studied the mathematics behind many reinforcement learning and neural network algorithms, only to realize that I had to read up on basic calculus and other subjects which were not included in the discrete mathematics course I took ten years ago. The work on the thesis has been stretched over a long period, mostly due to the fact that I work full time, which has left only evenings and holidays for working on the thesis. This would not have been possible, had it not been for the support of my loving wife Anja Pedersen, who has spent many evenings starring at my back while I have been working on the thesis. Neither

\(^1\)Can be downloaded from [http://leenissen.dk/fann/](http://leenissen.dk/fann/)
would it have been possible, if I did not have the support from my colleagues and
the management at BankInvest, who have allowed me to take time off from work in
the critical last weeks of writing, and who have been supportive even when I have
showed up at work with only a few hours of sleep behind me.

There are many people who I will thank for helping me accomplish the goals of
this thesis. I will not try to name them all here, but only name the most important.
I wish to thank:

**Michael L. Littman (Rutgers University),** who with his extensive knowledge
of the reinforcement learning field helped me confirm that the \(Q\)-SARSA(\(\lambda\))
algorithm was novel. Michael was always ready to answer my questions about
reinforcement learning. He guided me to many interesting articles, and I am
especially grateful that he forced me to read up on some of the model based
reinforcement learning algorithms.

**Richard S. Sutton (University of Alberta),** who also helped me confirm that
\(Q\)-SARSA(\(\lambda\)) was a novel algorithm.

**Marc G. Bellemare (McGill University) and François Rivest (Université de Montréal),**
who have allowed me to use their backgammon implementation
for this thesis, which has saved me months of work.

**Everybody who has read my thesis** and commented on my work. Sometimes
when you get buried in details you forget the full picture. Outside comments
are very valuable in these situations. I would especially like to thank Berit
Løfstedt and my wife Anja Pedersen, who provided me with many valuable
comments and who helped me keep focus on the main subject.

**Friends and family,** who have been patient and understanding when I have not
had time to see them, and who have sent me drinks on facebook when I did
not have time to party with them.
Contents

1 Introduction .................................................. 1
   1.1 Problem Description ...................................... 1
   1.2 Motivation ................................................. 3
   1.3 Goals ...................................................... 4
   1.4 Challenges ............................................... 5
   1.5 Contributions ............................................ 6
   1.6 Reading Guide ........................................... 7

2 Cascading Neural Networks ................................. 9
   2.1 Function Approximation ................................... 9
      2.1.1 Regression and Classification ......................... 9
      2.1.2 Function Approximation Algorithms .................. 10
   2.2 Artificial Neural Networks .............................. 11
      2.2.1 Artificial Neural Network Training ................... 11
      2.2.2 Training Algorithms .................................. 12
   2.3 Motivations for the Cascade-Correlation Algorithm .... 13
      2.3.1 The Local Minima Problem ............................. 14
      2.3.2 The Step-Size Problem ................................ 15
      2.3.3 The Moving Target Problem ............................. 16
   2.4 The Cascade Architecture ............................... 16
      2.4.1 The Cascade-Correlation Algorithm ................... 17
   2.5 Benefits and Drawbacks of Cascade-Correlation ........ 20
      2.5.1 Benefits of Cascade-Correlation ....................... 21
      2.5.2 Drawbacks of Cascade-Correlation ..................... 22
      2.5.3 Overcoming Drawbacks of Cascade-Correlation ........ 23
   2.6 The Cascade 2 Algorithm .................................. 25
   2.7 Cascading Neural Network Implementation ............... 27

3 Cascading Neural Network Test ............................. 29
   3.1 Literature Comparison .................................... 29
   3.2 Test Problems ............................................. 31
      3.2.1 Choosing Test Problems ............................... 31
      3.2.2 Test Problem Line-up .................................. 33
   3.3 Test Configuration ........................................ 34
   3.4 Test Results ............................................... 35
      3.4.1 Test Observations ..................................... 37
      3.4.2 Test Conclusion ....................................... 41

4 Reinforcement Learning .................................... 43
   4.1 The Reinforcement Learning Problem ...................... 43
      4.1.1 The Markov Property .................................. 45
      4.1.2 Markov Decision Processes ............................. 46
      4.1.3 The Optimal Policy \( \pi^* \) ............................ 46
      4.1.4 Finding an Optimal Policy from \( R_{s\to a} \) and \( R_{s\to a'} \) ... 49
4.2 Learning With or Without a Model ........................................ 51
4.2.1 Model-Based Learning ............................................... 51
4.2.2 Model-Free Learning .................................................. 51
4.2.3 Model-Free versus Model-Based .................................... 52
4.3 Exploration versus Exploitation ........................................... 52
4.3.1 The ε-greedy Selection ............................................... 52
4.3.2 Boltzmann-Gibbs Selection .......................................... 53
4.3.3 Max-Boltzmann Selection .............................................. 53
4.3.4 Optimism in the Face of Uncertainty ............................... 54
4.3.5 Directed Exploration .................................................. 54
4.3.6 Combining Selection Strategies ..................................... 56
4.4 Temporal Difference Learning ........................................... 57
4.4.1 Temporal Difference Prediction ..................................... 58
4.4.2 Off-Policy Q-Learning ............................................... 59
4.4.3 On-Policy SARSA-Learning ......................................... 59
4.4.4 Off-Policy versus On-Policy Learning ................................ 61
4.4.5 Q-SARSA Learning .................................................... 62
4.5 Eligibility Traces .......................................................... 63
4.5.1 n-step Return .......................................................... 64
4.5.2 λ-return ................................................................. 64
4.5.3 Eligibility Traces ........................................................ 64
4.5.4 The Q-SARSA(λ) Algorithm ........................................... 66
4.6 Generalization and Function Approximation .............................. 68
4.6.1 Function Approximation and Exploration ............................ 70
4.7 Model-Based Learning ..................................................... 72
4.7.1 Combining Model-Based and Model-Free Learning .................. 73
5 Reinforcement Learning and Cascading ANNs .......................... 77
5.1 Batch Training With Cache .............................................. 77
5.2 On-line Cascade-Correlation ............................................ 78
5.3 Incremental Training versus Batch Training ............................. 78
5.4 A Sliding Window Cache .................................................. 79
5.4.1 Eligibility Traces for the Sliding Window Cache ..................... 81
5.5 Neural Fitted Q Iteration .................................................. 82
5.5.1 Enhancing Neural Fitted Q Iteration ................................. 83
5.5.2 Comparing NFQ-SARSA(λ) With Q-SARSA(λ) ...................... 84
5.6 Reinforcement Learning and Cascading Networks ..................... 86
5.7 Reinforcement Learning Implementation ................................ 88
6 Reinforcement Learning Tests .............................................. 89
6.1 Reinforcement Learning Test Problems ................................ 89
6.1.1 The Blackjack Problem .............................................. 92
6.1.2 The Mountain Car Problem ......................................... 93
6.1.3 The Cart Pole Problem ............................................... 94
6.1.4 The Backgammon Problem .......................................... 95
6.2 Reinforcement Learning Configurations ................................ 96
6.3 Tabular Q-SARSA(λ) ..................................................... 98
6.3.1 Tabular Q-SARSA(λ) for Blackjack ................................. 99
6.4 On-line Incremental Neural Q-SARSA .................................. 106
6.4.1 Incremental Q-SARSA for Mountain Car .......................... 107
6.4.2 Incremental Q-SARSA for Cart Pole ................................ 108
6.5 Batch Neural Q-SARSA(λ) .............................................. 109
6.5.1 Batch Q-SARSA(λ) for Mountain Car .............................. 111
6.5.2 Batch Q-SARSA(λ) for Cart Pole ................................... 114
6.6 Cascading Neural Q-SARSA(λ) ........................................ 117
6.6.1 Cascading Q-SARSA(λ) for Mountain Car ......................... 119
6.6.2 Cascading Q-SARSA(λ) for Mountain Car Revisited .............. 124
Chapter 1

Introduction

This thesis explores how artificial neural networks (ANN) trained with the Cascade 2 algorithm can be combined with model-free reinforcement learning, and how this combination can be scaled to large problem sizes. Model-free reinforcement learning has historically been combined with neural networks with good results, but up until now, only a few people have tried combinations which include advanced neural network training algorithms such as the Cascade 2 algorithm.

1.1 Problem Description

Combining reinforcement learning with artificial neural networks is not new, and many successful applications of this combination has been documented in the literature. However, the algorithm which is usually used for training the neural networks in combination with reinforcement learning is a simple incremental algorithm, which is usually not the most optimal for normal neural network problems. More advanced neural network training algorithms exists, but the combination of reinforcement learning and advanced neural network training algorithms has one severe complication, which has prevented researchers from pursuing this combination. This complication lies in the fact, that while standard reinforcement learning methods such as Q-learning and SARSA learning are on-line incremental algorithms, the advanced neural network training algorithms are all off-line batch algorithms. This section will explain the basic idea behind the on-line reinforcement learning algorithms, and how they can be combined with incrementally trained neural networks. This explanation will clarify why it is more complicated to make the same combination with advanced neural network training algorithms.

Reinforcement learning is a learning method where an autonomous agent wants something, but where it does not know how to achieve this goal. In each step, the agent is informed of the current state of the environment, and it is given a choice of several different actions. Since the agent does not know how to achieve its goal, it tries to learn this by means of trial-and-error.

A simple example of a reinforcement learning problem is displayed in figure 1.1. Here the agent is a cart that moves on a track, with seven different positions, and the goal for the agent is to move to position 4 and stay there. In each step the agent will be given its current position (the state), and a choice of three different actions; move one position to the right, stay at the current position or move one position to the left.

The agent knows nothing of the environment, except for what it experiences and in the beginning it will not know which actions that are beneficial and which are not. It will not even know that it should choose to stay when it is in state 4. As
CHAPTER 1. INTRODUCTION

Figure 1.1: A simple reinforcement learning problem, with seven different states and three different actions; right, stay and left. The goal for the cart is to move to the green state 4 and stay there.

The agent tries different actions in different states, it will slowly build up knowledge about how profitable it is to take action $a$ in state $s$, and store this knowledge in a profit function $Q(s,a)$. After each step, the agent updates the $Q(s,a)$ value for the taken action, with the knowledge gathered from that step. When the most profitable action should be chosen in a given state, the agent compares the $Q(s,a)$ values for the three actions and choose the action with the highest $Q(s,a)$ value. Figure 1.2 shows the full learned $Q(s,a)$ function for the cart agent, where green squares represent positive $Q(s,a)$ values and red squares represent negative $Q(s,a)$ values.

Figure 1.2: The cart agent with at fully learned $Q(s,a)$ function, where green squares represent a positive $Q(s,a)$ value, and red squares represent a negative $Q(s,a)$ value. In each step the agent can take the current state, look up the $Q(s,a)$ value for the three possible actions and choose the most profitable one.

Since there are only seven states and three actions, the $Q(s,a)$ function can be maintained in a table for this simple example, but for more advanced problems this is not the case.

A more advanced problem, is the cart pole problem where a pole must be balanced on top of the cart, as displayed in figure 1.3.

Figure 1.3: The cart should balance the pole, by applying ten levels of left force, ten levels of right force or no force to the cart.

In this case the state is not only being represented by the position of the cart, it is also being represented by the velocity of the cart, the angle of the pole and the velocity of the pole. The actions are still right, stay and left, but in order to obtain more control, the right and left actions have been split into ten different levels of force, giving a total of 21 actions. Figure 1.4 shows the cart agent for the cart pole problem and here the $Q(s,a)$ function can not be represented by a table,
because all of the four state values are continuous. Instead the \( Q(s, a) \) function is represented by a black box, which takes the four state values and the action as an input, and returns the \( Q(s, a) \) value. When the most profitable action should be chosen, the agent simply compares the \( Q(s, a) \) values for the 21 actions and choose the action with the highest \( Q(s, a) \) value.

![Cart position, Cart velocity, Pole angle, Pole velocity, Black Box, Action, Q state, action](image)

Figure 1.4: Cart agent for the cart pole problem. The \( Q(s, a) \) function is represented by a black box, which takes the state and action as an input and returns \( Q(s, a) \) value.

An artificial neural network could be used to represent the black box, and the \( Q(s, a) \) values could be updated after each step with the knowledge gathered from taking that step, in the same way as a \( Q(s, a) \) function represented by a table. This approach was used by Tesauro (1995), when he implemented the famous TD-Gammon, which was the first backgammon playing program that was able to beat expert level players.

The simple updating of the \( Q(s, a) \) values in the neural network after each step, has proven very successful, and is still the primary method used for combining reinforcement learning with neural networks. This method uses a neural network training algorithm called incremental back-propagation, which is the simplest method of neural network training. Other more advanced neural network training algorithms exists, but none of these algorithms are designed for updating the \( Q(s, a) \) values after each step, as reinforcement learning algorithms requires. All of the advanced neural network training algorithms are off-line batch algorithms which require complete knowledge before training can begin. This knowledge is not available in the case of reinforcement learning, since the agent only acquires one piece of extra knowledge for each step it takes. Overcoming this obstacle and combining the advanced neural network training algorithm Cascade 2 with reinforcement learning is the main focus of this thesis.

### 1.2 Motivation

The incremental back-propagation neural network training algorithm which is normally combined with reinforcement learning was created by Werbos (1974) and later enhanced by Rumelhart et al. (1986). A few years later this algorithm was combined with reinforcement learning by Watkins (1989), an approach which was later made famous by Tesauro (1995). Since Rumelhart et al. (1986) many advanced neural network training algorithms have been developed, and these algorithms have shown far better results than the original back-propagation algorithm. Some of the most interesting training algorithms are the Cascade-Correlation algorithm (Fahlman and Lebiere, 1990) and its successor the Cascade 2 algorithm (Fahlman et al., 1996). What distinguishes these algorithms from the other advanced algorithms is that not only do they train the neural network, they grow the
network, starting with a small network. This feature is very appealing, since it removes the need for manually tuning the network size, and the Cascade-Correlation and Cascade 2 algorithms have shown very promising results.

Even though the advanced neural network training algorithms have been around for several years, they were not combined with reinforcement learning before Rivest and Precup (2003) combined reinforcement learning with Cascade-Correlation. Since then a few articles have been published, which explore different methods of combining Cascade-Correlation with reinforcement learning, and Bellemare et al. (2004) even tried the combination on the backgammon problem. The results from the different experiments have been mixed, and the experiment with backgammon failed entirely, which means that the combination have yet to prove its worth on large scale problems.

The most used neural network training algorithm in combination with reinforcement learning is the incremental back-propagation and another method which is also often used to represent the black box from figure 1.4 is CMAC (Albus, 1975). CMAC cannot represent as advanced $Q(s, a)$ functions as neural networks, but it is very stable, and have shown good results for many simpler reinforcement learning problems.

I believe that reinforcement learning with advanced neural network training algorithms have the potential of being far superior to reinforcement learning with incremental back-propagation or CMAC. There are several reasons for this belief, the primary being that the advanced algorithms usually perform better than incremental neural network training for normal reinforcement learning problems. Another reason is that advanced training algorithms usually have a more global view of the problem, and should therefore be able to more effectively use knowledge gathered over a longer period. However, the advanced algorithms are more difficult to combine with reinforcement learning and some of the potential may be lost due to limitations in the method used for combination.

For this thesis I wish to implement a combination which is able to produce results that can show the potential of the advanced neural network training algorithms. I especially wish to produce results which can show that the combination can scale to larger problem sizes, since the experiments with backgammon by Bellemare et al. (2004) suggested that the combination could not scale.

1.3 Goals

This thesis has three key focus areas: “Advanced neural network training algorithms”, “The standard model-free reinforcement learning algorithms” and “The combination of advanced neural network training and reinforcement learning”. This section will describe the scope of these three areas and the goals within the areas. These goals will be combined to form a main goal for the entire thesis.

**Advanced neural network training algorithms** come in many variations, and although other algorithms will be discussed, the Cascade 2 algorithm will be the primary focus, and the goal will be to explore the performance of the Cascade 2 algorithm in combination with reinforcement learning. The Cascade 2 algorithm will be given extra attention because it, unlike most other advanced algorithms, is able to grow the neural network from an empty network, which means that there is no need for manually tuning the size of the neural network.

**The standard model-free reinforcement learning algorithms** are the $Q(\lambda)$ and SARSA($\lambda$) algorithms. These algorithms have been used with great success for many reinforcement learning problems, and are the algorithms that are usually used in combination with neural networks. For this thesis my focus
1.4. CHALLENGES

will be these algorithms and variations of these, and the main measurement of performance will also be against variations of these algorithms. This does not mean that other algorithms will not be considered, but they will not be considered with the same weight. The reason for this is partly to keep the focus of the thesis more narrow, and partly because my goal is to improve on the most common combination of reinforcement learning and neural networks. This combination is more thoroughly investigated in literature than other combinations, and the results achieved from enhancing this combination will be easier to transfer to other combinations, than the other way around.

The combination of advanced neural network training and reinforcement learning is a challenging area, which have not been investigated thoroughly in the literature. For this thesis I will survey the existing combinations and use this survey to develop a more effective combination. When developing this combination two areas will receive extra focus: “Performance and scalability” and “Exploration and analysis”.

Performance and scalability goes hand in hand, but it must be remembered that scalability is the main reason for including neural networks in a reinforcement learning algorithm. For small and medium sized reinforcement learning problems there exists good model-based algorithm, which generally perform better than model-free algorithms. However, these algorithms do not scale well to larger problem sizes. A combination of the Cascade 2 algorithm with reinforcement learning can not be expected to perform better than the model-based algorithms for smaller problem sizes, but it can scale to very large problem sizes. For this reason, the primary goal will be to scale the combination to larger problem sizes, and the secondary goal will be to achieve performance that can compare to model-based algorithms.

Exploration and analysis is just as important as performance and scalability. The goal is to explore combinations of advanced neural network training algorithms and reinforcement learning, and analyze strengths and weaknesses of these combinations. The analysis is very important, because the combination with advanced neural network algorithms can be done in many different ways. It must therefore be an important goal to discuss how the combinations should be made for this thesis, and after these combinations have been tested, the evaluation must analyze the strengths and weaknesses, so that other researchers can use the results to create even better combinations.

The goals identified within the three key focus areas are combined to form the goal for the entire thesis:

My goal for this thesis is to explore and analyze how neural networks, trained with advanced neural network training algorithms such as Cascade 2 can be used to enhance the performance and scalability of the standard model-free reinforcement learning algorithms $Q(\lambda)$ and SARSA($\lambda$).

1.4 Challenges

There are many challenges when working with reinforcement learning and neural networks, but the main challenge of this thesis is the combination of reinforcement learning and advanced neural network training algorithms.
CHAPTER 1. INTRODUCTION

The reinforcement learning algorithm described in this chapter is an on-line algorithm which updates the $Q(s, a)$ values after each step. This on-line approach is a key element in reinforcement learning, because it allows the agent to learn while it is exploring the environment, and use this learned knowledge to achieve better results. For the cart pole problem, it can e.g. not be expected that the agent will stumble upon a $Q(s, a)$ function which will enable it to balance the pole for a longer period. It will start out by only being able to balance the pole for a couple of steps, and then slowly learn how to balance it for longer periods.

The advanced neural network training algorithms are batch algorithms, meaning that they need the entire training data-set, before they can train the neural network. The greatest challenge for this thesis is finding a good way of combining the batch neural network training algorithms with the on-line reinforcement learning algorithm. This challenge is even greater in the case of the Cascade 2 algorithm, because the algorithm grows the network from the gathered knowledge, and if the gathered knowledge is not sufficient when the network is grown, it may grow in an undesirable way.

Another challenge is scaling the algorithm towards larger problem sizes. It is not common to test reinforcement learning algorithms on large problem sizes and there exists no large standard problems which can be used as test examples. This challenge is further increased by the fact that there does not exist any successful examples of reinforcement learning combined with advanced neural network training algorithms for large problems.

This thesis addresses the challenges of combining reinforcement learning with advanced batch neural network training algorithms by applying a novel combinational method called the sliding window cache. This method is combined with modifications to the standard reinforcement learning algorithms and to the way that the Cascade 2 algorithm trains the neural network. This combination is tested on both medium sized standard reinforcement learning problems and the larger backgammon problem, in order to test the performance and scalability of the method. The tests show that not only is the method more effective than the standard incremental algorithm for smaller problems, it is also able to scale to large problems.

1.5 Contributions

This thesis provides the following contributions to the field of artificial neural networks, reinforcement learning and the combination of reinforcement learning and neural networks.

Artificial Neural Networks:

- A thorough description of the Cascade 2 algorithm and the mathematics behind it, which have not been described before.

- An implementation of the Quickprop, RPROP and Cascade 2 neural network training algorithms in the open source FANN library.

- A thorough classification of function approximation problems, which helps determine how function approximation algorithms should be tested in order to give a clear view of their performance.

- A thorough benchmark and comparison of the back-propagation, Quickprop, RPROP and Cascade 2 algorithms, which have been missing from the neural network literature.
1.6 Reading Guide

This thesis is written in English, although it is not my native language. This has posed a challenge for me, but since the thesis presents several contributions to the field of neural networks and reinforcement learning, I feel that it is important that it can be read by people outside Denmark. The thesis use italics to emphasize concepts such as the sliding window cache. However, to avoid all of the thesis being in italics, only concepts that have not been mentioned for a while is emphasized.

This thesis discusses many different aspects of neural networks and reinforcement learning. It is recommended that the reader has some prior knowledge of these aspects, or is willing to read some of the referred literature, in order to fully understand these discussions. However, all of the most important algorithms and theories have been explained in great detail, so the only prior knowledge which is required, is basic understanding of computer science and artificial neural networks. Readers with no prior knowledge of artificial neural networks are recommended to read appendix A first, as this will give a basic introduction.

The intended audience for this thesis is computer science students and teachers with interest in, but not necessarily knowledge of, artificial intelligence, reinforcement learning and neural networks. However, since the thesis provides several key contributions to the combination of reinforcement learning and neural networks, researchers with extensive knowledge of this field are also in the audience, but they are suggested to use the structure overview below or the table of context and go directly to the desired section.

Chapter 1 – Introduction: Introduces the problem, states the goal for the thesis and the contributions provided by the thesis.

Reinforcement Learning:

- A novel model-free reinforcement learning algorithm $Q$-SARSA($\lambda$), which combines $Q(\lambda)$-learning and SARSA($\lambda$) learning.

The Combination of Reinforcement Learning and Neural Networks:

- An implementation of the sliding window cache, which support the combination of the $Q$-SARSA($\lambda$) algorithm with the full set of training algorithms in the FANN library, including RPROP and Cascade 2.
- Several enhancements to the Neural Fitted $Q$ Iteration algorithm (Riedmiller, 2005), including support for the full $Q$-SARSA($\lambda$) algorithm, and combination with the Cascade 2 algorithm. The enhanced algorithm is named NFQ-SARSA($\lambda$).
- Implementation of the full NFQ-SARSA($\lambda$) algorithm with support for the full set of training algorithms in the FANN library.
- First structured benchmark to compare reinforcement learning combined with incremental back-propagation, batch RPROP training and the Cascade 2 algorithm.
- First combination of reinforcement learning and advanced neural network training, that can scale to larger problems.
Chapter 2 – Cascading Neural Networks: Explains the concept of function approximation, argues why neural networks and the Cascade 2 algorithm should be used in combination with reinforcement learning, and describes in great detail how the Cascade 2 algorithm grows its neural network.

Chapter 3 – Cascading Neural Network Test: Tests the performance of the Cascade 2 implementation, so as to give an idea of how well it will perform when it is later combined with reinforcement learning.

Chapter 4 – Reinforcement Learning: Describes the theory behind model-free reinforcement learning, and propose the $Q$-SARSA($\lambda$) algorithm which is a combination of the $Q(\lambda)$ and SARSA($\lambda$) algorithms.

Chapter 5 – Reinforcement Learning and Cascading ANNs: Describe how the cascading neural network architecture can be combined with the $Q$-SARSA($\lambda$) reinforcement learning algorithm, by means of the novel sliding window cache, and by means of the Neural Fitted $Q$ Iteration algorithm.

Chapter 6 – Reinforcement Learning Tests: Test the performance and scalability of different algorithms and combinations by testing them on smaller standard reinforcement learning problems, and the large backgammon problem.

Chapter 7 – Conclusion: Concludes the main achievements of the thesis.

Chapter 8 – Future Work: Proposes various directions that the work on combining reinforcement learning with advanced neural network training algorithms may go from here.

This thesis also contains a list of acronyms on page 251 and an index on page 252, and it is accompanied by a CD-ROM\textsuperscript{1}, which contains all the source code developed for this thesis along with this thesis as a pdf file. By using the source code on the CD-ROM, it should be possible to reproduce all the benchmarks by following the guidelines in appendix E.

\textsuperscript{1}The content of the CD-ROM can also be downloaded from \url{http://leenissen.dk/rl/}
Chapter 2

Cascading Neural Networks

A key aspect of model-free reinforcement learning is the $Q(s, a)$ function. The $Q(s, a)$ function provides an indication of how rewarding it will be to take action $a$ in state $s$. This could e.g. be an indication of how great the change is for winning a game of backgammon by making a specific move $a$ in a given board position $s$.

For small problems the $Q(s, a)$ function can be modelled by a simple tabular representation, but for larger problems a tabular representation is not feasible. When a tabular representation is not feasible, the $Q(s, a)$ function needs to be approximated.

This chapter introduces the concept of function approximation and describes how cascading neural networks can be used to provide good approximation of advanced functions.

2.1 Function Approximation

When we go to work in the morning we have to approximate how long the traveling time will be, in order to get to work in time. We do this by considering a number of factors like weather, traffic etc. and use our experience with traveling to work under these conditions to find out how long it will take today. The time it takes to get to work can be seen as a function of the different parameters that should be considered. Humans face these kinds of function approximation problems every day and solve them reasonably well, although we may sometimes fail and end up being late for work. Computers do not handle these kinds of problems as well and would much rather like precise functions, where they could e.g. calculate the time from a distance and an average speed.

When a software engineer faces these kinds of problems, he often resorts to using his own common sense, and would probably make a function that would increase travel time slightly if it is snowing or there is a lot of traffic. Often this can give good results, but often it is easier and more efficient to let the computer make the function approximation by looking at how much time we usually use to go to work under different conditions. Many different algorithms have been developed to approximate functions, some are restricted to special kinds of functions while others can approximate just about any function.

2.1.1 Regression and Classification

Function approximation problems can be split into two classes, classification and regression problems.
CHAPTER 2. CASCADING NEURAL NETWORKS

Classification problems are problems where the output is discrete, and they are often used to classify the input as belonging to one or more groups, hence the name. An example of a classification problem is the problem of recognizing handwritten numbers. In this case there are 10 unique classes that the input can belong to.

Regression problems are problems where the output is real valued. An example of a regression problem is the time to get to work. In this case the output is a real valued number representing e.g. minutes. Another example of a regression problem is the $Q(s,a)$ function, where the output is a real valued indication of how beneficial it will be to take action $a$ in state $s$.

Some function approximation problems might even consist of both regression and classification problems, but they could easily be split into separate regression and classification problems if that is desired.

Most general purpose function approximation algorithms can solve both kinds of problems, however, performance may be dependent on the problem type. Classification problems can be seen as a special case of regression problems, where the outputs are only allowed to take a discrete number of values. For this reason all algorithms that can be used for regression problems can also be used for classification problems while the opposite is not always true.

In theory there is not much difference between regression and classification problems, but in practice they often differ on the issue of how they are approximated. A classification problem will often have one binary output parameter for each of the classifications that the input can belong to. The job for the function approximator is to set an output if, and only if, the input belongs to the corresponding group. Because the output is binary, this can often be done very aggressively by only looking at a few features in the input. It is e.g. fairly easy to distinguish a handwritten “0” from a “1”, without including all information in the picture. Regression problems on the other hand often need to approximate some smooth, real valued, function where all the inputs might have influence on this value. In this case an aggressive approach is not desired, because the smooth nature of the function will be lost.

2.1.2 Function Approximation Algorithms

Function approximation algorithms work on the basis of learning-by-example, meaning that they are presented with examples of how the function evaluates (sets consisting of input and output values) and that they generalize from these examples in order to approximate the actual function.

The task for a function approximation algorithm is to approximate the output of a function for any valid input, after having seen input-output examples for only a small part of the input space.

A function approximation algorithm could be presented with the time it takes to get to work when there is hard rain and when there is no rain and it can then approximate the time it takes when there is light rain.

The examples that the function approximation algorithm uses for approximating the function are called training patterns and the entire training set is called a training data-set. Often the function approximation algorithm also has a validation data-set which is not used while approximating the function, but only for validating how well the solution generalizes. The process of approximating a function by looking at examples is called training.

Many different function approximation architectures and algorithms exists, but only a few is widely used in combination with reinforcement learning. The most
widely used is Cerebellar Model Articulation Controller (CMAC) (Albus, 1975; Glanz et al., 1991; Sutton and Barto, 1998), and Artificial Neural Networks (ANN). The main difference between these two architectures is the fact that ANNs can express non-linear functions, while CMACs can only express linear functions. The CMAC architecture is often used for reinforcement learning, because it is very fast and very stable. ANNs are, however, not as stable, and it is usually harder to get an ANN to work in combination with reinforcement learning. However, they are able to express more advanced functions, and they have been used with great success in some of the most successful reinforcement learning applications. These applications include the TD-Gammon backgammon program by Tesauro (1995) and helicopter control (Bagnell and Schneider, 2001).

For this thesis ANNs have been chosen as the function approximator, partly due to its prior success, and partly because the author of this thesis is also the creator and maintainer of the open-source ANN library Fast Artificial Neural Network Library (FANN)Nissen (2003). Specifically it has been chosen to use cascading neural networks, which have some advantages over traditional neural networks. The remainder of this chapter will focus on describing how a cascading neural network is trained and chapter 5 will focus on how it can be combined with reinforcement learning.

2.2 Artificial Neural Networks

Artificial Neural Network (ANN) is an architecture developed to mimic the way the neurons in the human brain work. The idea of an artificial neuron was conceived by McCulloch and Pitts (1943), but it was not until Werbos (1974) proposed the back-propagation algorithm that ANNs gained momentum.

The most widely used kind of ANNs is the multilayered feedforward ANN, which consists of layers of artificial neurons with an input and an output layer. The neurons are connected by connections which only go forward in between the layers. The back-propagation algorithm and most other related algorithms trains an ANN by propagating an error value from the output layer and back to the input layer while altering the connections on the way.

2.2.1 Artificial Neural Network Training

A multilayer feedforward neural network consists of neurons and connections. The neurons are located in layers and the connections go forward between the layers. In a fully connected ANN all neurons in one layer have connections to all neurons in the next layer. Figure 2.1 shows a fully connected ANN with bias neurons (see more about bias neurons in section A.1.2).

Each of the connections in an ANN has a weight associated with it. When an input is presented to the ANN, the input values are propagated along the connections and multiplied with the weights for the connections. In the neurons, all of the input connections are summed together and executed through an activation function (see section A.1.2), the output of this activation function is the output of the neuron. This eventually gives output values for the output neurons. If these values differ from the desired values, the ANN can be trained to minimize this difference. Appendix A gives a more thorough introduction to ANNs and how ANNs can be trained. It is advised that readers without any prior knowledge of ANNs and ANN training read this section before proceeding.

\footnote{FANN can be freely downloaded from http://leenissen.dk/fann/}
CHAPTER 2. CASCADING NEURAL NETWORKS

2.2.2 Training Algorithms

The dominating training algorithm for training ANN is back-propagation (see section A.1.3) and most other training algorithms are derivations of the standard back-propagation algorithm. There are two fundamentally different ways of training an ANN using the back-propagation algorithm:

Incremental training The weights in the ANN are altered after each training pattern has been presented to the ANN (sometimes also known as on-line training or training by pattern).

Batch training The weights in the ANN are only altered after the algorithm has been presented to the entire training set (sometimes also known as training by epoch).

When using only the basic back-propagation algorithm, the incremental training has a clear advantage because it learns faster and does not get stuck in a local optimum so easily (Wilson and Martinez, 2003). Batch training does, however, have a better global view of the training, so more advanced algorithms can be developed on the basis of batch training.

Many different algorithms have been developed on basis of batch back-propagation algorithm. Some of the most noticeable and effective are RPROP (Riedmiller and Braun, 1993) and Quickprop (Fahlman, 1988), but a number of other algorithms exist using momentum and variable step-size to speed up training. The Quickprop training algorithm is also the basis of the Cascade-Correlation and Cascade 2 algorithms which are covered in section 2.4.1, and which will be the primary training algorithm in this thesis.

Since training an ANN is simply a matter of adjusting the weights, many have viewed ANN training as an optimization problem, which can be solved by techniques used for general optimization problems. These techniques include simulated annealing (Kirkpatrick et al., 1987), particle swarm (Kennedy and Eberhart, 1995), genetic algorithms (Goldberg, 1989), Levenberg-Marquardt (More, 1977) and Bayesian techniques (Neal, 1996).

An approach which can be used in combination with these training algorithms is ensemble learning (Krogh and Vedelsby, 1995; Diettrich, 2000), which trains a number of networks and uses the average output (often weighted average) as the real output. The individual networks can either be trained using the same training samples, or they can be trained using different subsets of the total training set. A technique known as boosting (Schapire, 2001) gradually creates new training sets and trains new networks with the training sets. The training sets are created so that they will focus on the areas that the already created networks are having problems.
2.3. MOTIVATIONS FOR THE CASCADE-CORRELATION ALGORITHM

with. These approaches have shown very promising results and can be used to boost the accuracy of almost all of the training algorithms, but it does so at the cost of more computation time.

All of these algorithms use global optimization techniques, which means that they require that all of the training data is available at the time of training. For this reason these training algorithms can not be used directly in reinforcement learning, since reinforcement learning is on-line and requires that the ANN is trained while the data is generated. However, incremental training does not have these restrictions, since it only requires that one training pattern is available each time it adjusts the weights and can easily be combined with reinforcement learning. Luckily most of these global optimization techniques can be used in mini-batch training, which combines incremental and batch training by dividing the training data into small batches and train on these batches instead of only one large batch. The mini-batch algorithm is mentioned in the ANN FAQ (Sarle, 2002) and is empirically tested to perform better than standard batch back-propagation by Wilson and Martinez (2003). Rivest and Precup (2003) uses a derivation of the mini-batch algorithm in combination with reinforcement learning, which shows promising results, although the results of Bellemare et al. (2004) suggest that it might not scale to larger problems. The mini-batch algorithm has been implemented as a part of the reinforcement learning implementation for this thesis, and the results will be discussed further in chapter 6.

It is often difficult to determine the number of layers and hidden neurons that should be used in an ANN, and it is also difficult to determine the wide variety of parameters that can be adjusted for most ANN training algorithms. The need for hand tuning of the algorithms give rise to a number of dynamic algorithms which do not require that much tuning. One of these dynamic algorithms is optimal brain damage (LeCun et al., 1990), which alters the architecture of the ANN by removing connections. This results in a more compact and faster ANN which also often achieves better results, than the original ANN.

Another approach to dynamically altering the architecture of an ANN is to add connections and neurons through controlled growth. The algorithms that utilize this approach have the advantage that the size of the ANN need not be defined in advance and can therefore easier be used as general purpose function approximation. Parekh et al. (1997), Tenorio and Lee (1989), Fahlman and Lebiere (1990), Prechelt (1997) and Treadgold and Gedeon (1997) investigate several different approaches to growing ANNs. Parekh et al. (1997) investigates algorithms that builds networks consisting of Threshold Logical Units (TLUs) (see equation A.1.2), while Tenorio and Lee (1989) proposes the SONN algorithm which uses a simulated annealing approach to growing ANNs. Treadgold and Gedeon (1997) investigates an approach which utilizes the RPROP algorithm with different learning rates for different parts of the ANN. Fahlman and Lebiere (1990) proposes the Cascade-Correlation algorithm which will be described in greater detail in section 2.4 and Prechelt (1997) investigate algorithms that are all variants of the Cascade-Correlation algorithm. The Cascade-Correlation algorithm has shown very promising results, and is also the most widely used algorithm which uses controlled growth.

2.3 Motivations for the Cascade-Correlation Algorithm

The Cascade-Correlation algorithm has shown good results for several different problems, both with regards to generating compact ANNs and generating ANNs that provide accurate results. When Fahlman and Lebiere (1990) presented the
Cascade-Correlation algorithm, they demonstrated its power on the two-spiral problem, which is the problem of determining which of two interlocking spirals a point belongs to in a two dimensional image. This problem is particularly difficult to solve, because the problem is extremely non-linear. The problem is, however, very well suited for Cascade-Correlation, since each candidate neuron can focus on gaining correct results for a small part of the spiral, which will eventually give accurate results for the entire spiral. Fahlman and Lebiere (1990) also demonstrate Cascade-Correlation on parity problems, which share many properties with the two-spiral problem. Both the two-spiral problem and parity problems are classification problems (see section 2.1.1) which consists of artificial noise-free data. The limitations of the tests based on these two problems, indicates the need of other more thoroughly made tests. Ribeiro et al. (1997), Rivest and Precup (2003) and Littman and Ritter (1992) have tested the algorithm on other kinds of problems and these tests suggest that the Cascade-Correlation algorithm also perform well on real-world problems. The results of Batavia et al. (1996), do, however, show that not all problems are well suited for the Cascade-Correlation algorithm. Although tests of the Cascade-Correlation algorithm have shown mixed results, the algorithm still possessed some interesting properties, which is why I will investigate the algorithm further, to uncover its potential as a function approximator for reinforcement learning.

The Cascade-Correlation algorithm solves the problem of having to manually tune the ANN size, because it uses controlled ANN growth, but there are several other problems associated with the back-propagation training algorithm, which can lead to slow training or getting stuck in a local minimum. These problems are both evident in batch back-propagation and incremental back-propagation. Fahlman and Lebiere (1990) identifies the step-size problem and the moving target problem, but other problems can also be identified. The local minima problem is one such problem. This section describes these problems and section 2.5 describes how the Cascade-Correlation algorithm overcomes these problems.

### 2.3.1 The Local Minima Problem

The training of an ANN can be seen as an optimization problem, where the difference between the actual output of the ANN and the desired output of the ANN should be minimized. This difference is minimized by altering the weights in the ANN, which effectively gives an $N$-dimensional optimization space, where $N$ is the number of weights.

The back-propagation algorithm uses the gradient (sometimes referred to as slope) to determine which direction that is most beneficial, meaning that it will always go downhill in the optimization space, and that it will therefore always steer towards a local minimum. The difficult task for the algorithm is to step over all of the local minima and reach a global minimum.

A local minimum in the $N$-dimensional optimization space is a point where a small move in any direction will lead to a worse solution. An example of this in a one-dimensional optimization space can be seen in figure 2.2. An optimization space can include many local minima and ANN algorithms can easily get stuck in these minima. The local minimum can be viewed as a hole in the optimization space. Many of these holes are very small, but others are large. A simple example of a local minimum is often discovered when trying to approximate the XOR function. In this example the ANN will approximate the OR function by simply setting all weights to a high value, and thereby getting 3 out of the 4 solutions right. This local minimum can often be a large hole in the optimization space, since approximating the XOR function will require that some of the weight are shifted from their high value to a negative value.
2.3. MOTIVATIONS FOR THE CASCADE-CORRELATION ALGORITHM

Figure 2.2: An example of a local minimum. Here the standard back-propagation algorithm starting at the “Start Position” is very likely to get stuck in the local minimum, and hence never reach the global minimum.

Batch back-propagation suffers enormously from the local minima problem when a small step-size is used, since it has no way of escaping a local minima, where the radius of the hole is larger than the step-size. Incremental back-propagation however, does not suffer as much from this problem, since the steps that the incremental algorithm takes is only dependent of the gradient of a single training pattern. This allows the incremental algorithm to take several steps in a direction which is not the same as the gradient for the full training set, effectively allowing it to escape a local minimum. However, there is no guarantee that the incremental algorithm will succeed in escaping the local minimum.

Choosing a larger step-size can avoid some of these local minima, but if the step-size is chosen too large, the algorithm will also miss the global minima. More advanced training algorithms based on the back-propagation algorithm do not get stuck as easily in local minima, because they use a dynamic step-size, but other approaches to avoiding local minima can also be used.

2.3.2 The Step-Size Problem

Each step the back-propagation algorithm takes, is taken on the basis of the gradient. This gradient can tell the algorithm which direction that would be most beneficial, but it can not tell how large a step should be taken in that direction to reach a good solution. If the step-size is infinitely small, the algorithm will always reach a local minima, but it will take infinitely long time, and there is no way of knowing if the local minima is also a good global solution. If the step-size is too large, the ANN will not reliably reach a local minima, since the step can easily overshoot the local minima.

The step-size problem is evident both in the batch back-propagation algorithm and in the incremental algorithm, although it shows itself in different ways. In the batch back-propagation algorithm the problem shows itself much as described here, but in the incremental back-propagation algorithm the problem shows itself in a different way. Incremental back-propagation shows some of the same problems when the step-size is small, but since one step is taken for each input pattern, the incremental algorithm will generally be faster, and since each step is based on a new input pattern, the algorithm will have a tendency move jittery through the optimization space, and will not as easily get stuck in some of the small local minima. However, the problem of overshooting a good local minima is, even more evident in the incremental algorithm, since the jittery nature of the algorithm can make the algorithm miss a local minima even when the step-size is small.
2.3.3 The Moving Target Problem

In each epoch all the weights are altered according to the gradient of the corresponding connections, but each time a weight is altered, the outputs of the ANN are also altered, and so are the gradients for all of the other connections\(^2\). Since the gradients are only calculated once each epoch and all the gradients change each time a weight is changed, only the first weight change will be made using the correct gradient and all the remaining weight changes will be made on the basis of gradients that have changed since they were calculated. In large networks with many weights, the combination of all the independent weight updates can cause the final output of the ANN to move in an undesired direction. This problem is known as the moving target problem.

The problem of incorrect gradients is however only one part of the moving target problem. The other part is that since the weights are all changed independently of each other, they cannot cooperate. This inability to cooperate means that each weight will be prone to try to solve the same problem, even though an optimal solution would require that some weights focus on one problem while other focus on other problems. The inability to cooperate is enhanced by the fact that the gradients are incorrect.

One of the purposes of the random initial weights is to ensure that all weights do not pursue the same problem, but in many cases all of the weights will start to solve the most prominent problem (the problem that generates the largest error value), and when this is solved, they will all start to solve less prominent problems. The problem, however, is that as the weights starts solving less prominent problems, they will not solve the initial problem as well and they will then try to solve that again. This elaborate “dance” can go on for quite some time, until the weights split up, so that some solve the initial problem and others solve the other problems.

If this problem is looked at in the view of the optimization space, then what happens, is that the solution gets stuck in a local minima and in order to get out of the local minima, some of the weights will have to change while others remain focussed on the initial problem. In the optimization space this can be seen as a very narrow road aligned in a way so that most of the variables are not changed (or only changed a little) while others are changed a lot. On the road, the error value will be beneficial, while this will not be the case to each side of the road. In an ANN where all the weights work independently it can be very difficult to get the weights to navigate down such a road.

2.4 The Cascade Architecture

The cascade architecture was designed to avoid the local minima problem, the step-size problem, the moving target problem and to avoid having to define the number of layers and neurons up front. This section describes how the cascade architecture and the Cascade-Correlation algorithm functions, and section 2.5 describes how the architecture and algorithm can be used avoid the three problems. The cascade architecture consists of two algorithms:

A cascade algorithm which defines how neurons should be added to the neural network. The two most common algorithms is the Cascade-Correlation algorithm described in section 2.4.1, and the Cascade 2 algorithm described in section 2.6.

\(^2\)Except for the specific case where a weight to an output neuron is changed, since it has no effect on the gradients for the connections to the other output neurons.
2.4. THE CASCADE ARCHITECTURE

A weight update algorithm which is used to train the weights in the neural network and in the neurons that are added to the network. This algorithm was originally the Quickprop algorithm, but the implementation in this thesis supports both Quickprop and RPROP.

The key idea of the cascade architecture is that neurons are added to an ANN one at the time, and that their input weights do not change after they have been added.

The new neurons have input connections from all input neurons and all previously added neurons. This means that all of the hidden neurons in an ANN created by the cascade architecture will be located in single-neuron layers and that the ANN will be fully connected with short-cut connections. Short-cut connections are connections that skip layers and a fully connected ANN with short-cut connections is an ANN where all neurons have input connections from all neurons in all earlier layers including the input layer and output connections to all neurons in later layers.

2.4.1 The Cascade-Correlation Algorithm

The algorithm which introduced the cascade architecture is the Cascade-Correlation algorithm, which was introduced by Fahlman and Lebiere (1990).

The Cascade-Correlation algorithm starts out with an ANN with no hidden neurons as illustrated in figure 2.3 and figure 2.4. Figure 2.4 uses a more compact notation which is better suited for short-cut connected ANNs. The ANN has a bias neuron (see section A.1.2) and is fully connected with connections from all input neurons to all output neurons. The activation function \( g \) (see section A.1.2) in the original Cascade-Correlation algorithm was a hyperbolic tangent activation function (defined in equation A.1.4), but other functions could also be used. In the FANN library this function is referred to as the symmetric sigmoid function, because it closely resemble the normal sigmoid function with the only difference that it is in the range -1 to 1 instead of 0 to 1. This ANN is trained with a weight update algorithm like Quickprop or RPROP.

![Diagram of an ANN with no hidden neurons](image)

Figure 2.3: The initial network used in Cascade-Correlation training. All inputs are connected to all outputs and during initial training all weights are trained.

The Cascade-Correlation algorithm uses the cascade architecture to add new neurons to the ANN. Before the neurons are added they are trained, so that they can fulfill a productive role in the ANN. The neuron that is to be added to an ANN is called a candidate neuron. A candidate neuron has trainable connections to all input neurons and all previously added neurons. It has no direct connections to the output neurons, but it still receives error values from the output neurons. Figure 2.5 illustrates a candidate neuron which is being trained before it is added to the ANN illustrated in figure 2.4.

The cascade architecture freezes all input connections to a candidate neuron after it has been added to the ANN, which means that it is very important that
CHAPTER 2. CASCADING NEURAL NETWORKS

Figure 2.4: Same ANN as figure 2.3, but using another notation which is better suited for ANNs with short-cut connections. The values crossing the vertical lines are summed together and executed through the activation function “$g$”. All inputs are connected to all outputs and during initial training all weights are trained (marked by $\times$).

Figure 2.5: A candidate unit trained on the initial network is connected to the inputs, but not directly connected to the outputs although the error values in the outputs are used during training (visualized by a dotted line). During training of the candidate neuron only the inputs to the candidate is trained (marked by $\times$), while the other weights are kept frozen (marked by $\square$).

the candidate neuron is trained efficiently, and that it can fulfill a beneficial role in the ANN. A way of making sure that this happens is to train several different candidate neurons. Since the candidates are not dependent on each other they can be trained in parallel as illustrated in figure 2.6. The candidates will be initialized with different random weights in order to ensure that they investigate different regions of the optimization space. However, other methods to ensure this can also be utilized; The candidates can use different activation functions or can be trained by different training algorithms. The candidates can also use different parameters for the training algorithms allowing some of the candidates to use a small step-size and others to use a large step-size. Chapter 6 investigates how the cascade architecture performs when the candidates use the same activation function, and how they perform when different activation functions are used.

A candidate in the Cascade-Correlation algorithm is trained to generate a large activation whenever the ANN that it should be inserted into generates a different error value, than it does on average. This will allow the newly trained candidate to fulfill a role in the ANN that had not been fulfilled efficiently by any of the previously added neurons. The idea behind this approach is that the candidate will have a large activation whenever the existing ANN does not perform well, and that the ANN will be able to perform better once the candidate is installed, since its large activations can be used to combat the shortcomings of the original ANN.
2.4. THE CASCADE ARCHITECTURE

In order to generate this activation, the candidates input connections are trained to maximize the covariance $S$ between $c_p$, the candidate neurons value for pattern $p$, and $e_{k,p}$, the error (calculated by equation A.1.9) at output neuron $k$ for pattern $p$. $S$ is defined in equation 2.4.1, where $P$ is the training patterns, $K$ is the output neurons, $\overline{c}_p$ is the average candidate output over all training patterns $P$, and $\overline{e}_k$ is the average error at output neuron $k$ over all training patterns $P$.

$$S = \sum_{k=0}^{K} \sum_{p=0}^{P} (c_p - \overline{c})(e_{k,p} - \overline{e}_k)$$  \hspace{1cm} (2.4.1)

$S$ calculates the covariance for a candidate neuron and not the correlation as one might suspect from the name of the algorithm. Covariance and correlation are closely related, but if the correlation should have been calculated, the covariance should have been divided by the standard deviation of $c_p$ and $e_{k,p}$. Fahlman and Lebiere (1990) originally tried using the correlation instead of the covariance, but decided on using the covariance since it worked better in most situations.

Adjusting the input weights for the candidate neuron in order to maximize $S$, requires calculation of the partial derivative $\partial S/\partial w_{i,c}$ of $S$ with respect to the weight $w_{i,c}$, which is the weight for the connection from neuron $i$ to the candidate neuron $c$. $\partial S/\partial w_{i,c}$ is defined in equation 2.4.2, where $\sigma_k$ is the sign of the covariance for output unit $k$, $g'_i$ is the derived of the candidates activation function $g$ for training pattern $p$ and $o_{i,p}$ is the output from neuron $i$ for training pattern $p$.

$$\partial S/\partial w_{i,c} = \sum_{k=0}^{K} \sum_{p=0}^{P} \sigma_k (v_{k,p} - \overline{v})(g'_i o_{i,p})$$  \hspace{1cm} (2.4.2)

The partial derivative $\partial S/\partial w_{i,c}$ for each of the candidates incoming connections is used to perform a gradient ascent in order to maximize $S$. The weight update is made using a weight update algorithm. While the candidates are trained, all of the weights in the original ANN are frozen as illustrated in figure 2.6.

The candidates are trained until no further improvement is achieved, and when all of the candidates have been trained, the candidate with the largest covariance $S$ is chosen. This candidate will be installed into the ANN by freezing its input connections and making output connections to all output neurons, which is initialized with small random values. Figure 2.7 illustrates the resulting ANN after installing one of the candidates from figure 2.6 in the ANN.

When a new candidate has been installed into an ANN, the Cascade-Correlation algorithm once again trains all of the output connections using a weight update.
algorithm. If the ANN performs well enough, the training is stopped, and no more neurons are added to the ANN. If, however, the ANN does not perform well enough, new candidate neurons with input connections to all input neurons and all previously added neurons are trained (see figure 2.8).

When these new candidates have been trained one of them will be selected for installation into the ANN. This candidate will be installed as a single-neuron layer, as described by the cascade architecture, with connections to all previously added neurons and connections with random weights to the output neurons. An installation of one of the candidate neurons from figure 2.8 is illustrated in figure 2.9.

This process continues until the ANN performs well enough, and the training is stopped. The resulting ANN can be used as a function approximator just like any other ANN, and it can also be used to approximate the $Q(s,a)$ function for reinforcement learning.

### 2.5 Benefits and Drawbacks of Cascade-Correlation

The motivation for creating the Cascade-Correlation algorithm, was to overcome the local minima problem, the step-size problem and the moving target problem. Section 2.5.1 will concentrate on how the Cascade-Correlation algorithm handles these problems, while section 2.5.2 will concentrate on which drawbacks the algorithm has and how they can be handled. The discussion about benefits and drawbacks leads
2.5. BENEFITS AND DRAWBACKS OF CASCADE-CORRELATION

Figure 2.9: New candidate neurons are inserted with input connections to all previously inserted neurons and output connections to all output neurons. This effectively builds an ANN with single-neuron layers and short-cut connections. After each candidate has been inserted into the ANN, the output connections are trained (marked by $\times$).

to the Cascade 2 algorithm, which is a Cascade algorithm based on the Cascade-Correlation algorithm. The Cascade 2 algorithm is described in section 2.6 and section 2.7 describes how this algorithm is implemented in the thesis.

2.5.1 Benefits of Cascade-Correlation

The local minima problem, the step-size problem and the moving target problem are all problems that exists for standard incremental and batch back-propagation algorithms. The more advanced algorithms based on batch back-propagation, like Quickprop and RPROP avoids the local minima problem and the step-size problem by using a dynamic step-size, but they still suffer from the moving target problem. This section will first describe how Quickprop and RPROP used dynamic step-sizes to avoid the local minima problem and the step-size problem, it will then go on to explain how the Cascade-Correlation algorithm can avoid all three problems, partly by using Quickprop and RPROP as weight update algorithm, and partly by means of the cascade architecture.

The Quickprop Algorithm

The Quickprop algorithm (Fahlman, 1988) computes the gradient just like batch back-propagation, but it also stores the gradient and the step-size from the last epoch. This pseudo second-order information allows the algorithm to estimate a parabola and jump to the minimum point of this parabola.

This dynamic step-size allows the Quickprop algorithm to move effectively through the optimization space with only a limited risk of getting stuck in small a local minimum, and when the global minimum is near, the algorithm will only use small steps, and hence reach the minimum. This effectively means that the Quickprop algorithm to a large degree avoids the local minima and the step-size problems.

The parabola is estimated independently for each weight in the ANN and the jump to the minimum point is also made independently for each weight. This independence between the updates in the individual weights is necessary because the algorithm would otherwise be forced to only update one weight for each epoch, or do a much more advanced estimate of the parabola. The argument for this independence is an assumption that a small change in one weight only alters the gradient of the other weights by a relatively small factor. This is also correct, but since all of the weights in an ANN change each epoch, the gradient for each weight is changed many times during an epoch. This means that the Quickprop algorithm
will suffer enormously from the moving target problem, when the steps along the parabola is large, while the problem will not be so dominant when the step-size is smaller.

The RPROP Algorithm

The RPROP algorithm (Riedmiller and Braun, 1993) also uses pseudo second-order information about the step-size and gradient, but instead of trying to estimate a point on a parabola, it simply looks at the sign of the two gradients. If they have the same sign, then the algorithm is still walking down the same hill, and the step-size is increased. If the gradients have different signs, this is an indication that the algorithm have overshot the local minima, and the RPROP algorithm reverts the weight to the previous position and decreases the step-size.

This approach has the advantage that each weight can adapt to a problem independent of the other weights and the size of the gradient, since only the sign of the gradients is used, which in tests have shown very promising results.

Like the Quickprop algorithm, the RPROP algorithm also avoids the local minima problem and the step-size problem by means of the dynamic step-size, but it still suffers from the moving target problem. The moving target problem is, however, not as prominent a problem, since the RPROP algorithm will decrease the step-size whenever the target moves.

The Cascade-Correlation Algorithm

The Cascade-Correlation algorithm avoids the local minima problem and the step-size problem by using Quickprop or RPROP as the weight update algorithm, but it also applies another strategy which further helps to avoid the local minima problem. When the candidates are trained, several candidates are trained in parallel, which effectively means that all of the candidates will need to get stuck in a local minimum in order for the entire training to get stuck. This effect can also be seen in the tests in section 3.4, where it is very seldom that the cascade architecture gets stuck in a local minimum.

The moving target problem can be addressed by only allowing some of the weights in the ANN to change at any given time. The Cascade-Correlation algorithm utilizes this approach in an extreme way, where at any given time, either the output weights or the weights of a candidate neuron, are allowed to be altered. Using this approach the weights can easily move to fill a beneficial role in the ANN, while the remaining weights are frozen. This approach effectively cancels out many of the problems concerning the moving target problem, but it also introduces some problems of its own which will be discussed in section 2.5.2.

2.5.2 Drawbacks of Cascade-Correlation

Cascade-Correlation has several drawbacks which may prevent good learning and generalization.

Covariance training has a tendency to overcompensate for errors, because the covariance objective function $S$, optimizes the candidates to give large activations whenever the error of the ANN deviates from the average. This means that even when only a small error occurs, the output of the candidate is optimized to be very large. This feature makes Cascade-Correlation less suited for regression problems, since it lacks the ability to fine-tune the outputs. However, classification problems do not suffer from this overcompensation in the same degree, since the output values which should be reached in classification problems are the extreme values. In many situations classification problems
can even benefit from overcompensation, because it forces the outputs to the extreme values very quickly.

**Deep networks** generated by Cascade-Correlation, can represent very strong non-linearity, which is good for problems which exhibit strong non-linearity. Many function approximation problems do, however, exhibit a high level of linearity. If these problems are solved using a non-linear solution, they will most likely provide very poor generalization since the linear nature of the problem will not be visible in the solution. This problem is referred to as **over-fitting** in normal ANNs where it is also evident for ANNs which have too many layers and neurons.

**Weight freezing** in Cascade-Correlation is a very efficient way to overcome the moving target problem. Weight freezing does, however, pose some problems of its own, as explained below.

If a candidate is located in a local minimum, and it does not manage to escape the minimum during candidate training, the ANN will grow without achieving better performance. As the ANN grows, it will be harder to train the output connections and it will be harder to add new neurons. If the algorithm at some point escapes the local minimum, there will be a lot of frozen connections which cannot be used, and which will make it more difficult to train the resulting ANN. *Kwok and Yeung (1993)* describes how this is often a problem for the first few candidates which are added to the ANN.

These drawbacks of Cascade-Correlation are usually outweighed by the advantages, and all in all Cascade-Correlation is an effective training algorithm for ANNs. However, sometimes Cascade-Correlation is outperformed by other algorithms. An example of Cascade-Correlation being outperformed, is the *Autonomous Land Vehicle in a Neural Net* (ALVINN) problem, where experiments by *Batavia et al. (1996)* show that Quickprop outperforms both the Cascade-Correlation and the Cascade 2 algorithm (described in section 2.6), for this particular problem.

### 2.5.3 Overcoming Drawbacks of Cascade-Correlation

The drawbacks can sometimes be overcome by altering the Cascade-Correlation algorithm or the parameters for the algorithm. Altering an ANN algorithm and determining if the alteration is an improvement can however be a bit of a problem since some ANN algorithms work well for some problems while other ANN algorithms work well for other problems. The descriptions of how to overcome the problems here, should therefore not be seen as instructions on how the Cascade-Correlation algorithm should be altered, but rather instructions on how the algorithm (or its use) could be altered if these problems arise.

**Covariance training** can be replaced by direct error minimization, making the Cascade-Correlation algorithm perform better for regression problems. Direct error minimization is known as the Cascade 2 algorithm (described in section 2.6). *Prechelt (1997)* have benchmarked the Cascade 2 and the Cascade-Correlation algorithms. The results showed that Cascade 2 was better for regression problems while Cascade-Correlation was better for classification problems. When ANNs are used in reinforcement learning, the problems that need to be solved are regression problems, so it would seem that Cascade 2 is a better choice in this case.

**Deep networks** can be overcome by simply allowing candidate neurons to be placed both in a new layer, and as an extra neuron in the last hidden layer.
Baluja and Fahlman (1994) experiment with a variation of this approach, in which candidates allocated in the last hidden layer receive a small bonus when determining which candidate should be chosen. This approach shows that the depth of the ANN can be reduced dramatically, but unfortunately the generalization skills of the ANN is not improved.

The problem of over-fitting during Cascade-Correlation might not be as huge a problem as first suggested, which is partly due to the fact that the Cascade-Correlation algorithm uses a patience parameter (see Fahlman and Lebiere (1990)) as a form of early stopping during training, which stops the training when the error has not changed significantly for a period of time. This enables the training of the candidates and the training of the output connections to be stopped before too much over-fitting occurs. However, the patience parameter is not used for the main loop of the Cascade-Correlation algorithm, so over-fitting can still occur if too many candidates are added to the ANN. Squires and Shavlik (1991) have made experiments with the patience and they conclude that it both helps generalization and execution time.

Over-fitting can still be a problem for the Cascade-Correlation algorithm, especially if it is allowed to grow too much. A simple way of reducing the chance of over-fitting is to train using more training patterns, since the algorithm will then see a more precise picture of the function it must approximate.

Weight freezing was originally used by Fahlman and Lebiere (1990) as a way to overcome the moving target problem, but later research by Squires and Shavlik (1991), Kwock and Yeung (1993) and Treadgold and Gedeon (1997) have questioned this effect. The results are inconclusive but shows that for some problems weight freezing is not desired. Three approaches can be used when the input weights should not be frozen, one which have been suggested before, and two novel approaches:

- Not freezing the inputs at all, and simply train the entire ANN when the Cascade-Correlation algorithm describes that only the connections to the output neurons should be trained.
- I suggest not freezing the inputs entirely, but keep them cool, meaning that a very small step-size is used for the inputs while the connections to the output neurons use a larger step-size. (As far as I know, this is the first time this solution have been proposed). This approach is inspired by the Casper algorithm by Treadgold and Gedeon (1997), which is a constructive algorithm that adds new neurons to the ANN, and then trains the entire ANN with different step-sizes for different parts of the ANN. Experiments with the Casper algorithm shows very promising results.
- I also suggest doing a two part training where the output connections are trained first, and the entire ANN is trained afterwards. This solution allows the newly added candidate to find a proper place in the ANN before the entire ANN is trained. (As far as I know, this is the first time this solution have been proposed). This approach is used for experiments with combining the Cascade 2 algorithm with reinforcement learning in chapter 6, and the results show that this can especially be beneficial for problems where many candidate neurons are added.

Since the ANN described in this thesis should be used for reinforcement learning, which uses regression problems, it would be beneficial to use the Cascade 2 algorithm instead of the Cascade-Correlation algorithm. Weight freezing might also
be a problem, so experiments with different approaches to weight freezing can also be beneficial. For this thesis the Cascade 2 algorithm has been implemented as described in section 2.6, and as part of the reinforcement learning implementation, the two part training has been implemented to avoid weight freezing.

## 2.6 The Cascade 2 Algorithm

The discussion in section 2.5.3, suggest that the Cascade 2 algorithm should be used when a regression problem should be solved. Since estimating the $Q(s,a)$ function is a regression problem, I will investigate the Cascade 2 algorithm further.

The Cascade 2 algorithm was first proposed and implemented by Scott E. Fahlman, who has also proposed and implemented the Cascade-Correlation algorithm. He wrote an article (Fahlman et al., 1996) in collaboration with D. Baker and J. Boyan about the algorithm, but the article was never published and it has not been possible to locate the article. Fahlman did however communicate a bit with Lutz Prechelt, when Lutz was writing (Prechelt, 1997), and Lutz included a little information about the Cascade 2 algorithm in his article. Most of the information, including all equations are, however, obtained by inspecting the source code developed by Fahlman (written in LISP) and C ports of this code. Since no published material (except for the source code) exists documenting the Cascade 2 algorithm, this section will document the most important aspects of the algorithm.

The Cascade 2 algorithm is a modified Cascade-Correlation algorithm, where the training of the candidates has been changed. Candidates in the Cascade 2 algorithm have trainable output connections to all of the outputs in the ANN. These connections are still only used for propagating the error back to the candidates, and not for propagating inputs forward in the ANN. The difference lies in the fact that for the Cascade 2 algorithm, these output connections are trained together with the inputs to the candidates.

A candidate is trained in order to minimize the difference between the error of the output neurons and the input to the output neurons, received from the candidate. Note that this is done while the main ANN is frozen, just like normal Cascade-Correlation and that the input from the candidate to the output neurons is only used for training the candidate and not used for calculating the error at the output neurons. When the candidate is inserted into the ANN, its output weights are inverted, so that the candidates contribution to the output neurons will help minimize the error.

The difference between the error of the output neurons and the input from the candidate to these neurons $S_2$ is defined in equation 2.6.1, where $e_{k,p}$ is the error at output neuron $k$ for output pattern $p$ and $w_{c,k}$ is the weight from the candidate to output neuron $k$. This means that $c_p w_{c,k}$ is the input from the candidate neuron $c$ to the output neuron $k$.

$$S_2 = \sum_{k=0}^{K} \sum_{p=0}^{P} (e_{k,p} - c_p w_{c,k})^2$$  (2.6.1)

In order to minimize $S_2$, the partial derivative $\partial S_2/\partial w_{c,k}$ of $S_2$ with respect to the weight $w_{c,k}$ needs to be calculated. $\partial S_2/\partial w_{c,k}$ is defined by equation 2.6.2.

---

3 I tried to contact all of the authors of the article but I was not able to locate the article and they were not willing to discuss the Cascade 2 algorithm.

4 The original LISP code is not available on Fahlman’s site anymore, so I have posted it on the site for the FANN library [http://leenissen.dk/fann/cascade2.lisp](http://leenissen.dk/fann/cascade2.lisp). However, the C port is still available at [http://www.cs.cmu.edu/~sef/sefSoft.htm](http://www.cs.cmu.edu/~sef/sefSoft.htm).
\[ \frac{\partial S^2}{\partial w_{c,k}} = -2 \sum_{p=0}^{P} (c_p w_{c,k} - e_{k,p}) c_p \] (2.6.2)

When a candidate is inserted into the ANN, the output weights of the candidate are inserted with inverted sign. The idea behind this approach is that if \( S^2 \) is sufficiently low, the candidate will cancel out the error of the ANN. This idea relies on a linear activation function in the output neurons.

The error \( e_{k,p} \) for an output neuron \( k \) at output pattern \( p \) is given in equation 2.6.3, where \( y_{k,p} \) is the actual output of the neuron and \( d_{k,p} \) is the desired output of the neuron.

\[ e_{k,p} = d_{k,p} - y_{k,p} \] (2.6.3)

\( y_{k,p} \) can be calculated by equation 2.6.4, where \( n \) is the number of connections going into output neuron \( k \), \( w_{j,k} \) is the weight going from neuron \( j \) to \( k \), \( x_{j,p} \) is the output at neuron \( j \) for pattern \( p \) and \( g_k \) is the activation function used at neuron \( k \).

\[ y_{k,p} = g_k \left( \sum_{j=0}^{n} w_{j,k} x_{j,p} \right) \] (2.6.4)

If \( g_k \) is a linear activation function, then \( y_{k,p} \) is reduced to the sum, as seen in equation 2.6.5.

\[ y_{k,p} = \sum_{j=0}^{n} w_{j,k} x_{j,p} \] (2.6.5)

When the candidate \( c \) is inserted into the ANN, the new \( y_{k,p} \) will be calculated as shown in equation 2.6.6, which can be reduced to equation 2.6.7 by using the original \( y_{k,p} \) from equation 2.6.5. Here \( w_{c,k} \) is the weight from the candidate \( c \) to the output neuron \( k \), before this connection is inverted and \( -w_{c,k} \) is the inverted weight which is actually inserted into the ANN.

\[ y_{k,p}^{\text{new}} = w_{c,k} c_p + \sum_{j=0}^{n} w_{j,k} x_{j,p} \] (2.6.6)

\[ y_{k,p}^{\text{new}} = y_{k,p} + w_{c,k} c_p \] (2.6.7)

The new error \( e_{k,p}^{\text{new}} \) will then be calculated using the new output \( y_{k,p}^{\text{new}} \), as shown in equation 2.6.8, which in turn can be transformed into equation 2.6.11.

\[ e_{k,p}^{\text{new}} = d_{k,p} - y_{k,p}^{\text{new}} \] (2.6.8)

\[ e_{k,p}^{\text{new}} = d_{k,p} - (y_{k,p} + w_{c,k} c_p) \] (2.6.9)

\[ e_{k,p}^{\text{new}} = d_{k,p} - y_{k,p} - w_{c,k} c_p \] (2.6.10)

\[ e_{k,p}^{\text{new}} = e_{k,p} - w_{c,k} c_p \] (2.6.11)

Equation 2.6.11 shows that if a linear activation function is chosen at output neuron \( k \), then the error after inserting the candidate \( c \) to the ANN will be shifted by \( w_{c,k} c_p \). Since the candidate is trained to minimize the difference between \( e_{k,p} \) and \( w_{c,k} c_p \), then the squared sum of all \( e_{k,p}^{\text{new}} \) will approach zero as \( S^2 \) approaches zero.
2.7 Cascading Neural Network Implementation

The neural network in this thesis should be used to solve regression problems in reinforcement learning and since the Cascade 2 algorithm has shown better results for regression problems than the Cascade-Correlation algorithm (Prechelt, 1997), it will be more beneficial to use this algorithm. However, to the best of my knowledge, no implementation of the Cascade 2 algorithm in a neural network library exists. The Cascade 2 implementations which do exist will require a bit of work before they can be used as a function approximators for a reinforcement learning implementation. Even when this work is done, it will be necessary to use another ANN library, if the results are to be compared to other ANN training algorithms. I have decided to implement the Cascade 2 algorithm, partly to avoid this clutter and partly because I feel that the process of implementing this central algorithm will provide an insight to how the algorithm works, which would be difficult to acquire elsewhere.

I have decided to implement the Cascade 2 algorithm in the FANN (Nissen, 2003) library. This thesis is based on the FANN library version 1.1.0, which provides a basic neural network implementation.

For this thesis the following additions have been made to the FANN library:

- **Short-cut connections** which is vital for the implementation of the Cascade 2 algorithm.

- **Batch back-propagation** which is used as a basis for more advanced back-propagation algorithms.

- **The Quickprop algorithm** which is the weight update algorithm used for the original Cascade 2 algorithm.

- **The iRPROP$^-$ algorithm** which is a modification of the original RPROP algorithm, proposed by Igel and Hsken (2000). This implementation is used as an alternative to Quickprop for the weight update algorithm for Cascade 2.

- **Several new activation functions** which is used to create candidates for the Cascade 2 algorithm with different activation functions.

- **The Cascade 2 algorithm** which will be the basis for the reinforcement learning implementation for this thesis. The implementation only allows new neurons to be created in separate layers and the implementation does not provide any solutions to problems that might occur due to weight freezing. Although solutions to the weight freezing problem have been implemented in when the Cascade 2 algorithm is used in combination with the reinforcement learning implementation.

When candidates are trained during the Cascade 2 algorithm, they are trained so that they can fulfill a role in the ANN which is not yet fulfilled by the existing neurons. It is not possible prior to training the candidates, to determine which neuron will be best suited for this task. It is, however, possible to train many different candidates at the same time in the hope that one of these candidates will fulfill a beneficial role. If all of the candidates that are trained resemble each other, the chance of all of these getting stuck in the same local minimum is larger, than if the candidates do not resemble each other. For this reason it is desired that the candidate neurons do not resemble each other too much. The obvious method of making sure that the candidates do not resemble each other is by changing the initial weights. My implementation takes this a bit further by allowing the candidates to
have different activation functions, and for the candidates to have different steepness parameters (see A.3) for their activation functions.

The Cascade 2 algorithm implemented in the FANN library can use all of the implemented activation functions for the candidates and for the output neurons, but as described in section 2.6 a linear activation function is preferred for the output neurons. The implementation does not utilize the caching functionality which is suggested in Fahlman and Lebiere (1990), although this functionality would be able to speed up the training. The reason that this functionality was left out is that the functionality would not be able to make the training better, but only be able to make it faster. Implementing different training algorithms and activation functions had higher priority than implementing the cache functionality.

The additions that have been made to the FANN library is released with version 2.1.0 of the FANN library, and all tests have also been made on the basis of this version of the library. For instructions on how to replicate the tests, please see appendix E.
Chapter 3

Cascading Neural Network Test

The Cascade 2 implementation should be compared to results from the literature, in order to confirm that it is able to obtain similar results. When the Cascade 2 implementation is used in combination with the reinforcement learning implementation described in section 5.7, the Cascade 2 implementation should use RPROP and Quickprop as weight update algorithms. It is therefore important to determine how well these combinations perform compared to other algorithms. It is also important to determine how the parameters for the algorithms should be set in order for them to give the best results, so that very little tweaking of the ANN parameters needs to be done when testing the reinforcement learning implementation. The main focus of this thesis is, however, not on the efficiency of these algorithms, but rather their combination with reinforcement learning. The test should therefore focus on this concern. Section 3.1 will compare the Cascade 2 implementation to results from the literature, while the remainder of this chapter will focus on testing the implementation on a variety of test problems.

3.1 Literature Comparison

Since no articles have been published which directly concern the Cascade 2 algorithm, it is hard to say how it compares to other algorithms. Prechelt (1997) compares the Cascade 2 algorithm neck-to-neck with the Cascade-Correlation algorithm, on several different problems and concludes that the Cascade 2 algorithm is better for regression problems, and the Cascade-Correlation algorithm is better for classification problems. However, the benchmarks by Prechelt (1997) do not compare the Cascade 2 algorithm to other algorithms.

The Cascade 2 implementation in the FANN library includes several features which were not included in the tests made by Prechelt (1997), like the ability to include several different activation functions, and the ability to include iRPROP− as weight update algorithm. It is important that these features are compared to each other, to other algorithms and to published results from the literature.

The problems that the Cascade-Correlation algorithm was originally tested on (Fahlman and Lebiere, 1990), have not been used to test the Cascade 2 algorithm. These problems are noise-free classification problems, which should be especially well suited for the Cascade-Correlation algorithm, and not as well suited for the Cascade 2 algorithm. These problems will make a challenging test, that will be able to put the Cascade 2 implementation into a broader context.
Fahlman and Lebiere (1990) uses the N-Input parity problem, which is an excellent problem for literature comparison, because it is very easy to specify. Many of the other problems, like the two-spiral problems have the disadvantage, that you cannot know for sure which exact data Fahlman and Lebiere (1990) used for training. However, the N-Input parity problem have other problems which makes it ill-suited as the sole problem used for general benchmarking. There is no way of telling how well an algorithm generalizes when benchmarking on the N-Input parity problem, because the training data consists of the entire solution domain, furthermore the Cascade-Correlation algorithm has an advantage because the problem is a noise-free classification problem.

When Fahlman and Lebiere (1990) tested the 8-Input parity problem using the Cascade-Correlation algorithm, they used 357 epochs on average to generate ANNs which could solve the problem. The ANNs that were generated using the Cascade-Correlation algorithm, were generated with 4-5 hidden neurons. The same problem has been tested with four different Cascade 2 configurations, in order to compare the results to that of the Cascade-Correlation algorithm. The average results after five runs with each configuration can be seen in table 3.1. The “Single” configurations use 8 candidate neurons who all use a symmetric sinus activation function with a steepness parameter of 1 (see A.3), and the “Multi” configurations use 80 candidate neurons with 10 different activation functions and 4 different steepness parameters. The “Single” and “Multi” configurations are combined with both Quickprop and iRPROP as weight update algorithms.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Hidden Neurons</th>
<th>Average Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cascade-Correlation</td>
<td>4-5</td>
<td>357</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>1</td>
<td>62</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>1</td>
<td>38</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>1</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 3.1: The average number of hidden neurons and epochs of the four Cascade 2 configurations, compared to the Cascade-Correlation averages reported in (Fahlman and Lebiere, 1990)

Table 3.1 clearly shows that all of the Cascade 2 configurations perform better than the Cascade-Correlation algorithm. They produce much more compact networks, and they use less epochs to train. All of the configurations only use one hidden neuron, which is a huge improvement compared to the 4-5 neurons for the Cascade-Correlation algorithm. The best performing configuration is the “C2 Quickprop Single”, which only needs 38 epochs on average.

There can be several different reasons as to why the Cascade 2 algorithm performs better than the Cascade-Correlation algorithm. Various implementation details could account for the improved performance, but it seems like the major factor is actually the choice of activation function. Fahlman and Lebiere (1990) uses a sigmoid activation function for the output neuron and a Gaussian activation function for the hidden neurons. The Cascade 2 configurations use a linear activation function for the output neuron, and for the hidden neurons they either use multiple activation functions, or stick with a single symmetric sinus activation function. The symmetric sinus activation function in the FANN library, gives a periodic result between -1 and 1 and it performs extremely well for this particular problem. In comparison, using the symmetric sigmoid activation function and the iRPROP training algorithm generates networks with 5-8 hidden neurons and uses 1017 epochs on average. It is a surprise that the configuration with 8 candidates perform better.
than the configuration with 80 candidates. I believe that this is due to the fact that
the symmetric sinus activation function with a steepness of 1 performs very well,
and with 8 of these candidates the “Single” configurations manage to perform better
than the “Multi” configurations, which only have 2 candidates with this activation
function and steepness.

3.2 Test Problems

The comparison to the results by Fahlman and Lebiere (1990) provides an indication
that the Cascade 2 implementation performs well, for one particular problem. This
problem is, however, very far from the problem that the ANN should approximate
when it is combined with reinforcement learning. For this reason a more thorough
comparison of the implemented algorithms must be made. The comparison should
focus on the regression problem that the ANN should solve when combined with
reinforcement learning, but in order to provide a wide basis for comparison other
problems should also be considered.

3.2.1 Choosing Test Problems

When comparing different function approximation algorithms, one of the largest
problems is finding good data-sets to use for comparison. By nature some algorithms
are better at solving some problems while other algorithms are better at other. It
is not easy to determine why one algorithm is better than the other for a specific
problem, and for this reason it is even more difficult to determine which problems
are best suited for comparing function approximation algorithms.

Many of the existing comparisons between function approximation algorithms have
been made by comparing only a few different algorithms (Orr et al., 2000; Blanzieri,
1998) and often these algorithms have been closely related (Prechelt, 1997). More often the comparisons that have been made between algorithms have
been made by the person who have constructed one of the algorithms (Riedmiller and Braun, 1993; Fahlman and Lebiere, 1990). These comparisons may be valid and objective,
but there is still a chance that the author has used more time fine-tuning the pa-
rameters for his training algorithm, than he has for the competing algorithms or
that he has chosen a problem that his algorithm is particularly well-suited for. For
this reason one should be careful when using these comparisons as benchmarks for
the algorithms.

In order to make comparison of function approximation algorithms easier several
repositories of data-sets exists (Blake and Merz, 1998; Prechelt, 1994) and
some software suites for evaluating performance of function approximation algo-
rithms have also been produced, Data for Evaluating Learning in Valid Experi-
ments (DELVE) by Neal (1998) being the most prominent. DELVE does not seem
to be actively maintained, but it does offer an excellent basis for comparison of
function approximation architectures and algorithms. The algorithms compared in
the DELVE repository combined with other larger (Tan and Gilbert, 2003) and
smaller (Orr et al., 2000; Blanzieri, 1998; Prechelt, 1997; Riedmiller and Braun,
1993; Fahlman and Lebiere, 1990) comparisons, gives a scattered view of func-
tion approximation architectures and algorithms, which suggests that more research
within this area would be beneficial.

For this thesis I will try to determine some of the criteria that should be con-
sidered when comparing function approximation algorithms, in order to make the
comparison of the implemented ANN training algorithms as accurate and objective
as possible. Data-sets for function approximation problems can be separated into
several different groups, where there is a good possibility that an algorithm which
performs well for one problem in a group will also perform well for other problems in the same group. These groups can be used when determining which data-sets should be used for comparing ANN algorithms. A number of characteristics identify a group:

**Classification / Regression** The problem which should be solved, can either be a classification or a regression problem (see section 2.1.1).

**Linear / Nonlinear** The problem can express different levels of linearity and non-linearity.

**Small / Large** The problem can consist of a small or a large number of inputs and outputs.

**Noiseless / Noisy** The data-set can either be noisy or noiseless. Noise can be input or output data which are not exact (or missing), but it can also be parts of the problem space which is under- or over-represented in the data-set.

**Synthetic / Natural** The problem can either be a naturally occurring problem, or a created synthetic problem where the formula for the solution is known in advance.

These five characteristics can help split data-sets into a number of groups. The three first characteristics are especially interesting, because they identify the nature of the problem which needs to be solved. The fourth characteristic (noise) does not refer to the problem, but to the given data-set. It is often important that an algorithm can look beyond the noise and approximate the problem which lies below. The fifth characteristic does not always say anything about the nature of the problem or the training data, since both synthetic and natural problems can express the same characteristics. Often synthetic and natural problems do, however, differ in a number of ways. Synthetic data-sets can sometimes represent the complete problem space (the XOR problem space is e.g. covered by only four data-patterns), which means that training with this data-set will not tell anything about the generalization capabilities of the training algorithm. Synthetic problems have the advantage that it is often possible to make several different problems with different characteristics, from the same base problem (8-parity and 13-parity is e.g. derived from the XOR problem). This can be used to test how algorithms perform when the complexity of the problem is increased. Natural problems often include much noise and the distribution of the data-sets in the problem space is not as uniform as for synthetic problems. The problem space itself is also often much less uniform for natural problems, where some parts of the problem space can be very linear, other parts can be very nonlinear. These differences which often exists between natural and synthetic problems means that it is not always possible to reproduce good results from synthetic problems on natural problems. Since the main goal of ANNs is to be used on natural problems, this raises the question of whether synthetic problems should be used for testing ANNs.

When Lutz Prechelt created his PROBEN1 set of data-sets (Prechelt, 1994), he decided to include only natural data-sets, because of the problems with synthetic data-sets. PROBEN1 includes both classification and regression problems with both a small and a large number of input and outputs, they are, however, all natural and hence it is not possible to say much about their degree of linearity or their noise level. I do not believe that only including natural problems, is enough when comparing function approximation algorithms, because many of the benefits from using synthetic problems will be lost. However, only using synthetic
problems like Fahlman and Lebiere (1990) is not a good solution either. When testing the implemented algorithms both synthetic and natural of problems should be tested. The FANN library has a set of benchmarking problems which consists of a mix of synthetic and natural problems. Some of these problems have been taken from PROBEN1, some from DELVE and others have been taken from literature. These problems will be used when testing the iRPROP−, Quickprop and Cascade 2 training algorithms.

Some of the five characteristics like e.g. “Classification / Regression” represent a distinct classification into two groups of problems. Other characteristics like e.g. “Small / Large” are more loose and it can be difficult to determine which group a problem belongs to. The question may be raised of whether there should be two or more groups. Even if only two groups are used to represent these characteristics, 32 different groups will be needed in order to represent all combinations of the different characteristics.

Traditionally not much work has been done in identifying problems and data-sets, which fit nicely into each of these 32 groups. This has lead researchers to test new algorithms on only a few problems, which in turn means that comparing different algorithms by only looking at the published material can be very difficult. When Scott E. Fahlman published the Cascade-Correlation architecture (Fahlman and Lebiere, 1990) he only tested the algorithm on two problems and both of these problems were synthetic classification problems with zero noise, a high level of non-linearity and a small number of input and output neurons, which means that even though he used two different problems, he only tested one of the 32 groups. The Cascade 2 implementation has been compared to one of these problems in the literature comparison, but it needs to be tested on a broader range of the 32 groups.

### 3.2.2 Test Problem Line-up

The FANN library has a set of 16 data-sets, which will be used when comparing the different algorithms. These problems are summarized in table 3.2.

The Abelone, Census-house, Bank-32fm, Bank-32nh, Kin32-fm and Pumadyn-32fm data-sets are taken from DELVE (Neal, 1998) and the Diabetes, Gene, Mushroom, Soybean, Thyroid and Building data-sets are taken from PROBEN1 (Prechelt, 1994). The Parity8 and Parity13 problems are classical neural network problems, which are derived from the XOR problem. These problems are often referred to as N-parity problems, and was used when Fahlman and Lebiere (1990) introduced the Cascade-Correlation architecture. The two-spiral problem was also used in Fahlman and Lebiere (1990). The Robot problem is the problem which the FANN library was initially designed to solve, this is a real world problem of finding a white line in an image. The inputs are pre-processed image information and the outputs are information about the position and direction of the line (Nissen et al., 2003).

The problems have been selected in order to give as diverse a benchmarking configuration as possible, but they do not cover all of the 32 groups mentioned in section 3.2. All of the data-sets have had their inputs and outputs scaled to be in the 0 to 1 range, which can make comparison against literature difficult, but since the literature for comparison is very sparse, I do not consider this a problem. Almost all of the problems have test data-sets, which can be used for testing the performance of the trained ANN. However, the parity problems do not have test data-sets, since their training data-sets cover all of the available data patterns.

The Cascade 2 algorithm will build an ANN from scratch, and does therefore not need to have any hidden layers defined in advance. The other training algorithms do, however, need to have the hidden layers defined. This poses a problem, since it is not easy to define the optimal number of hidden layers and neurons. If these
3.3 Test Configuration

The purpose of this test is to determine how well the RPROP, Quickprop and Cascade 2 implementations perform compared to each other and compared to other ANN training algorithms. Ideally the implementations should be tested against a wide range of other algorithms and implementations, unfortunately no standard repository of implementations to test against exist, and creating these kinds of benchmarks from scratch is a very time consuming task. Instead I have chosen to make a complete set of benchmarks comparing the three algorithms to other algorithms in the FANN library and comparing them against two other neural network libraries. The two external libraries is included to provide a bias free comparison, which can reveal any weaknesses that the FANN library might have. The algorithms which will be used in the benchmarks are:

**Cascade2 RPROP Single:** The Cascade 2 algorithm with iRPROP\(^{-}\) as the weight update algorithm. 8 candidates are used, who all use the symmetric sinus activation function and a steepness parameter of 0.5 (see A.3).
3.4. **Test Results**

For each of the 16 training problems, each of the 10 training algorithms have been given 4 training cycles lasting 7 minutes each. During the training, the mean square error (MSE) and other key parameters are logged. The logs for the 4 cycles are combined to generate a graph showing how the average MSE of the training data and testing data evolves during the 7 minute span. The average values are also used to generate tables summarizing the algorithms performance on each of the problems. The graphs and tables for each of the 16 problems are displayed in appendix F. For each of the training algorithms, the tables show three values for the training data and three values for the test data. The first value is the best average MSE obtained during the 7 minute span. The second value is a rank from 1 to 10, where 1 is...
given to the algorithm with the best MSE, and 10 is given to the algorithm with the worst MSE. The last value is a percentage value indicating how far the MSE for the individual algorithm, is from the best MSE obtained by any of the algorithms. This percent is calibrated so that the algorithm with rank 1 will get a percentage of zero, and the algorithm with rank 10 will get a percentage of 100, while the other algorithms are given a percentage value based on a linear function of their MSE. The three values for the test data is also MSE, rank and percentage, only here the values are based on the results of the test data.

When comparing ANN algorithms, the result which is usually presented is the number of epochs, that the algorithms have been trained for. This value is good when comparing algorithms with results from literature, since factors like processor speed and various optimizations does not have any influence on the result.

If two different algorithms do not use the same amount of processing to perform each epoch, the number of epochs might not be the best result to present. Batch training does *e.g.* use less processing power than incremental training, because it only updates the weights once each epoch. When comparing these two algorithms, the actual time used for training is a better measurement parameter, since this measurement will take more factors into account. When comparing two algorithms within the same library, it must be assumed that the two algorithms are equally optimized, and I do not feel that epochs should be used in this case. Also when only comparing epochs, the Cascade 2 algorithm is not given a penalty for training many candidates at the same time, although this can be very time consuming.

For these reasons I have chosen to only look at the time factor, and neglect the number of epochs. The net effect of this decision is that the different algorithms are not given the same amount of epochs to train. A side effect of this, is that it is not always the same algorithm which is given the most amount of epochs. This is due to the fact that some algorithms and implementations are faster on small ANNs and training sets, while others are faster on larger ANNs and training sets. Also since the ANN trained with the Cascade 2 algorithm gets bigger all the time, the Cascade 2 algorithm will use more time on each epoch, as more neurons are added to the ANN. To take a closer look on this phenomena, two problems have been selected:

- The diabetes problem which as default have 8 inputs, 4 neurons in the hidden layer, 2 output neurons and 384 training samples.
- The mushroom problem which as default have 125 inputs, 32 neurons in the hidden layer, 2 output neurons and 4052 training samples.

The number of epochs which was trained during one of the tests is displayed in table 3.3. The figure clearly shows that the number of epochs that a particular implementation can reach is very much dependent on the problem that it must solve. I will not delve too much into why some algorithms are faster than other, but I will say that the FANN library is highly optimized for large ANNs at the expense of small ANNs, which explains why “Lwnn Incr.” is faster than “Incremental” for diabetes, and slower for mushroom.

As mentioned in section 2.7, the Cascade 2 implementation does not include a caching functionality, which should be able to speed up the training. This fact means that the Cascade 2 implementation could be even faster, and that the number of epochs executed by this algorithm could be enhanced further.

Another value which is not examined during the benchmark, is the number of neurons generated by the Cascade 2 algorithm. This value is very important when it is critical that the resulting ANN should be able to execute very fast, but it has no effect on the raw quality of the ANN. If the ANN should be optimized to be as small as possible, then it will sometimes be desirable to stop the training before the
Table 3.3: The number of epochs which was reached by each of the training algorithms within one of the 7 minute runs, for the diabetes and mushroom problems.

7 minutes have lasted, so as to make a tradeoff between quality and size. Since no such tradeoff has been made during the benchmarks it will not make much sense to report this value. Section 3.1 which compares the Cascade 2 implementation with published literature, however, compares both number of neurons in the final network and the number of epochs needed to generate the network.

### 3.4.1 Test Observations

Table 3.4 shows the average values of all of the tables in appendix F. These averages can be used to make observations, which can reveal strengths and weaknesses of the different configuration. It is, however, important to remember that these averages can hide local differences, since some configurations give better results for some kinds of problems, while other configurations give better results for other kinds of problems. In order to obtain the full picture, the tables in appendix F should be examined in conjunction with table 3.2 and table 3.4. For the sake of simplicity I will mostly look at the averages here.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Diabetes Epochs</th>
<th>Mushroom Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 RPROP Single</td>
<td>49235</td>
<td>10756</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>32105</td>
<td>3202</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>45149</td>
<td>9738</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>27529</td>
<td>6765</td>
</tr>
<tr>
<td>iRPROP</td>
<td>544222</td>
<td>4705</td>
</tr>
<tr>
<td>Quickprop</td>
<td>633124</td>
<td>4897</td>
</tr>
<tr>
<td>Batch</td>
<td>631229</td>
<td>4986</td>
</tr>
<tr>
<td>Incremental</td>
<td>607722</td>
<td>2429</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>1279081</td>
<td>1649</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>148663</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 3.4: The average values of all the training runs from appendix F

The Best Performing Configuration

It is easy to see that “C2 RPROP Single” obtains the best results during training, but it is however not as easy to see which configuration that obtains the best test results. “Lwnn Incr.” has the best MSE and percent value, while “iRPROP”
has the best rank. “C2 RPROP Single” and “C2 RPROP Multi”, however, also produce good results for rank and percentage. To establish which configuration gives the best overall results, a closer look at what the three different numbers express must be taken. The MSE is the value that is usually looked at when comparing training algorithms, but since this is an average MSE, it has the disadvantage that a single bad result, can have a large effect on the average. The rank eliminates this problem, but introduces another problem. When only looking at the rank, it is impossible to see whether there was a huge difference between number one and two, or the difference was insignificant. The percentage eliminates some of this uncertainty, but it introduces yet another problem. If one configuration performs very bad for a problem, then all of the other problems will seem to perform very well. Removing this problem from the benchmark suite could completely change the average percentage values for the other problems, and could even change their relative positions.

It is clear to see that the best test results must be found between the four configurations “C2 RPROP Single”, “C2 RPROP Multi”, “iRPROP−” and “Lwnn Incr.”. These four configurations have been summarized in table 3.5.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>MSE</th>
<th>Rank</th>
<th>Rank</th>
<th>Rank</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.034905</td>
<td>2.69</td>
<td>2.63</td>
<td>2.75</td>
<td>51.69</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.035158</td>
<td>2.44</td>
<td>2.38</td>
<td>2.50</td>
<td>47.23</td>
</tr>
<tr>
<td>iRPROP−</td>
<td>0.038363</td>
<td>2.38</td>
<td>2.50</td>
<td>2.25</td>
<td>53.79</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.028963</td>
<td>2.50</td>
<td>2.50</td>
<td>2.50</td>
<td>42.49</td>
</tr>
</tbody>
</table>

Table 3.5: The average values of the four configurations which performed best for test data. In this table the rank and percent is only calculated on the basis of the four configurations.

Table 3.5 clearly shows that the difference in rank and percent between the four algorithms is not very large, furthermore it shows that if you only look at the regression or classification problems, then the difference is still very small, although it seems that “C2 RPROP Single” falls a bit behind. When looking at the individual tables in appendix F it is revealed that all of the four configurations achieve close to optimal results for the mushroom and parity-8 problems, when these two problems are removed from the equation, then “C2 RPROP Multi” gets an average rank of 2.36, “Lwnn Incr.” gets 2.43, while “iRPROP−” gets 2.57 and “C2 RPROP Single” gets 2.64. These two problems seems to be the primary reason as to why “iRPROP−” gets so good results for the rank while MSE and percent falls a bit behind, it seems that “iRPROP−” is a bit better at getting close to a local optimum than the other configurations, while it does not perform as well overall.

Before forming any conclusions on the basis of this analysis I would like to give some attention to the selection of the benchmark problems, because this test shows exactly how important the selection of benchmark problems is. When the results are as close as they are in this test, then exchanging even a single benchmark problem with another could completely shift the picture. If e.g. the abelone problem was exchanged with a problem where “C2 RPROP Single” performed best, “Lwnn Incr.” second, “iRPROP−” third and “C2 RPROP Multi” performed worst, then the average rank of all of the four configurations would be exactly the same. With this fact in mind, it is impossible to give a clear answer to the question: Which configuration gives the best results for test data? When only looking at the four benchmark problems, it is however possible to say that the performance of “Lwnn Incr.” and “C2 RPROP Multi” seems to be a bit better than the two other con-
3.4. TEST RESULTS

figurations. “Lwnn Incr.” gets a better average MSE value, while “C2 RPROP Multi” gets a better average rank. The difference in percent is only 0.20%, which is insignificant.

When comparing these two configurations, on the 16 problems, it can be seen that the “C2 RPROP Multi” configuration performs better than “Lwnn Incr.” for 8 of the problems, and “Lwnn Incr.” performs better than “C2 RPROP Multi” for the remaining 8 problems. Removing the mushroom and parity-8 problem from the equation does not alter the results, since “Lwnn Incr.” performs best for mushroom and “C2 RPROP Multi” performs best for parity-8.

When it is clear that the two configurations perform equally well on average, the next question must be: Which kinds of problems is the one configuration better at, and which kinds of problems is the other configuration better at? Again there is no clear picture, they perform equally well on classification and regression problems, “Lwnn Incr.” has a slight edge in natural problems while “C2 RPROP Multi” has a slight edge in synthetic problems, but there is no clear result regarding size, linearity and noise, so the difference observed between natural and synthetic problems might just be a coincidence. Although “Lwnn Incr.” and “C2 RPROP Multi” perform equally well on test data, it must be kept in mind, that “C2 RPROP Multi” performs far better on train date. This seem to indicate that “C2 RPROP Multi” is generally better than “Lwnn Incr.”, although “C2 RPROP Multi” does have a tendency to over-fit.

The very accurate comparison made between these ten configurations, gives a good picture of how the configurations compare on this particular configuration of problems. The benchmarks do not display the full picture of how the individual algorithms perform. There are several different reasons for this:

- All of the different problem groups mentioned in section 3.2 are not present.
- The number of hidden layers and neurons may not be optimal for some of the problems, giving the Cascade 2 algorithm an unfair advantage, since they start with no hidden layers.
- The parameters for some of the algorithms might not be optimal, giving these algorithms unfair disadvantages.
- The implementations of the individual algorithms might not be perfect, and some algorithms may have been implemented to be faster than others.

This set of benchmarks does, however, give much better basis for comparing these algorithms than most of the earlier published comparisons. The changes made to the FANN library in order to facilitate these benchmarks, also make future comparison with other implementations of training algorithms very easy, and there is some hope that researchers might use this suite when presenting new algorithms, making the comparison with existing algorithms more objective.

When comparing the configurations, all of the configurations are given 7 minutes to reach their full performance. This gives an advantage to highly optimized implementations. However, looking at the graphs in appendix F reveal that many of the configurations reach their full potential way before the 7 minutes has passed, showing that the speed of the individual implementations is actually not that important in this comparison.

Comparing the Cascade 2 Configurations

It has already been established that the two best Cascade 2 configurations is the “C2 RPROP Single” and “C2 RPROP Multi” configurations, which would also indicate
that it is better to use the iRPROP$^-$ training algorithm than the Quickprop training algorithm, but it is more difficult to determine whether “Single” or “Multi” is the best choice.

When comparing the “Multi” configuration and the “Single” configuration, it can be seen that the “Single” most often get the best results for training data, while the “Multi” performs a bit better on the test data. On this basis, it is very difficult to determine whether it is better to add candidates with a single activation functions, or it is better to add candidates with several different activation functions. In this situation it should be mentioned that the activation function which is chosen for the “Single” configuration is very important. The benchmarks show very promising results for the symmetric sinus activation function, but the same results could not be reached using the standard symmetric sigmoid activation function. It is uncertain why the sinus activation function is better for these benchmarks, but (Sopena et al., 1999) suggest that using a periodic activation function is often superior to using a monotonic activation function.

When comparing the RPROP and the Quickprop versions of the Cascade 2 configurations, it is easy to see that RPROP gets far better results than the Quickprop versions. For the non-cascade versions of the iRPROP$^-$ and Quickprop algorithms, the same picture is visible. The Quickprop configuration is actually one of the worst performing configurations, which is a surprise. Had it not been for the very positive results with the Quickprop algorithm in the literature comparison, I would have been led to believe that there was a problem with the Quickprop implementation. The parameters for the Quickprop algorithm are the same for both the benchmarks and the literature comparison, and these parameters are also the parameters suggested by Fahlman (1988) when he presented the Quickprop algorithm. The main difference between the literature comparison and the benchmarks is that the literature comparison use an output span between -1 and 1, while the benchmarks use a span between 0 and 1. For the literature comparison, using the -1 to 1 span improved the performance, but it is not possible to say whether the same performance gain can be achieved for the benchmark problems.

Comparing FANN With External Libraries

There are two different external libraries, the Lightweight Neural Network Library (Rossum, 2003) and Jet’s Neural Library (Heller, 2002). Both of these libraries use incremental training, which mean that they should perform comparable to the FANN implementation of incremental training.

The three incremental implementations do not perform equally well, the Lwnn implementation perform best while the Jneural implementation perform worst. One of the reasons that the Jneural implementation performs worse than the two other implementations, is that Jneural is by far the slowest implementation of the three. The difference between the performance of the Lwnn and the FANN implementation cannot be caused by difference in execution speed alone, since they have comparable execution speeds, although Lwnn is a bit faster on average. The difference could be caused by difference in parameters, but the benchmarks try to use the same parameters for all of the implementations, so this should not really be a problem.

This raises the question: Why do two implementations of the same algorithm give different results? The answer lies in all the small details that goes into implementing a neural network and the algorithms that is used in it. These details include how to initialize the weights, how bias neurons are used and whether momentum is used. It also includes minor details and tricks which the implementations might use to avoid flat spots or too high weights. All of these details make different implementations of the same algorithms give different results. This is a problem when comparing algorithms, which in effect means that questions need always be raised.
whenever an article is released claiming one algorithm is better than another. It should always be remembered, that what is really being compared is the individual implementations. Another problem with these kinds of articles is that they often only compare a few problems, and that the parameters for the algorithms are often highly tuned for them to perform well on these particular problems. These articles do, however, often report some numbers which can be used to compare different algorithms, so that one article can be directly compared to another. Section 3.1 makes a comparison between the Cascade 2 implementation and numbers from the literature. This comparison in combination with the benchmarks provide a much better view of the performance of the implementation than either of the two would be able to do separately.

3.4.2 Test Conclusion

The purpose of this test was to see how well the Cascade 2 implementation performed compared to other training algorithms, and to determine which configuration of the Cascade 2 algorithm that performs best. The results from this test should be used to determine how Cascade 2 is best combined with reinforcement learning. Overall the Cascade 2 implementation combined with the iRPROP weight update algorithm performs very well. It clearly outperforms Fahlman and Lebiere (1990) implementation of Cascade-Correlation and it clearly gets the best results for the training data when compared to other algorithm configurations. The results for the test data in table 3.4 is, however, not as convincing. The results for “C2 RPROP Multi” are still good, but when compared to the training results they are a bit disappointing. There is an indication that generalization does not work as well for Cascade 2 as it does for the other configurations. This might be a problem when combining Cascade 2 with reinforcement learning, but it is too soon to say.
Chapter 4

Reinforcement Learning

Function approximators, such as artificial neural networks, learn an optimal behavior by looking at examples of optimal behavior (training data-sets). This approach is very useful when examples of optimal behavior are available. However, for many problems such examples do not exist since no information about the optimal behavior is known.

Reinforcement learning methods learns optimal behavior by trial-and-error, which means that it does not require any information about the optimal behavior in advance. This property makes reinforcement learning very interesting in an artificial intelligence perspective, since it is not bounded by the examples that are given to it. A function approximator can generalize from the examples that it is provided, but it can never learn to be better than the provided examples. This means that if a function approximator learns to play backgammon by looking at examples from an expert player, the function approximator can potentially learn to be just as good as the expert, but it can never learn to be better than the expert. A reinforcement learning algorithm does not have this limitation, which means that it has the potential to be better than the expert player. The advantages of this approach is obvious, but there is a significant drawback in the fact that it is much harder to learn optimal behavior when nothing is known about optimal behavior in advance.

This chapter will explain the principles of the reinforcement learning problem and deliver details of some of the most popular algorithms used to solve the problem, along with a few more advanced topics. Much of the fundamental theory in this chapter is based on Sutton and Barto’s book (Sutton and Barto, 1998) and the surveys (Kaelbling et al., 1996; Kelle, 2003; Keerthi and Ravindran, 1995). Functions, parameters and variables use the names and notations from Sutton and Barto (1998) wherever possible, since this notation is widely used in reinforcement learning literature.

4.1 The Reinforcement Learning Problem

The reinforcement learning model consists of an agent and an environment. At a given time $t$, the environment provides a state $s_t$ to the agent and the agent performs an action $a_t$ based on $s_t$. After the agent has performed the action $a_t$, the environment is updated to provide a new state $s_{t+1}$ to the agent. In addition to the state, the environment also provides a numeric reward $r_{t+1}$, which is an immediate reward or punishment for selecting the action $a_t$ in the given state $s_t$. The reinforcement learning model is summarized in figure 4.1. Which action an agent chooses in a given state, is determined by the agents policy $\pi$. The purpose
of any reinforcement learning method is to modify the policy in order to maximize the long-term (or sometimes short-term) reward.

Formally the reinforcement learning model uses discrete time steps $t$ and consists of a discrete set of states $S$, for each of these states $s \in S$ a discrete set of actions $A(s)$ exists. The agent chooses actions based on the policy $\pi$. There are generally two kinds of policy functions; the stochastic policy function, where $\pi(s, a)$ is the probability of the agent choosing action $a$ when it is in state $s$, and the deterministic, where $\pi(s)$ is the action $a$ which is chosen when the agent is in state $s$. The stochastic policy function can easily be made deterministic by simply choosing a greedy strategy, meaning that:

$$\pi(s) = \arg \max_{a \in A(s)} \pi(s, a) \quad (4.1.1)$$

often the action selected by a stochastic policy function, is however also referred to as $\pi(s)$, to ease notation. The agent only looks at the current state when deciding which action to choose, so any previous actions or states are not taken into account. The environment is stationary, in a manner of speaking. Stationary does not mean that taking action $a$ in a given state $s$ always yields the same result, but it means that the probability of getting to state $s'$ and receiving a reinforcement of $r$ by choosing action $a$ in a given state $s$ is always the same. A reinforcement learning problem can be episodic, meaning that some of the states in $S$ are goal states, and that the agent needs to be restarted whenever it reaches a goal state. Other reinforcement learning problems are non-episodic, meaning that $S$ does not contain any goal states. Many reinforcement learning algorithms does not distinguish between episodic or non-episodic problems, while others like e.g. Monte Carlo prediction only works for episodic tasks.

The formal reinforcement learning model somewhat limits the applications in which reinforcement learning can be used, but still many real life problems can be converted to fit this formal model:

- If a problem requires that the previous steps are remembered, then the state can be altered so that it includes this information.
- If a problem requires a real valued state parameter, like e.g. a speed, then this parameter can be split up into discrete values.
- Real valued actions can be made discrete in the same way.
- If an environment is not completely stationary, then the state can be modified to include this non-stationary part, making the environment stationary. Lets say that the agent is a robot, which observes its environment using a set of infrared sensors, which means that the state is the input from these sensors.
The sensors work perfectly when the ceiling light is turned off, but when ceiling light is turned on the sensors pick up that light and work less than perfect. In this case the state can be altered to include information about whether the ceiling light is on or off, making the environment stationary. If the probability that the ceiling light is on or off is always the same, then there is actually no need to alter the state, since the environment is already considered stationary, although it may still be difficult to learn a good policy.

Most theoretical reinforcement learning work is based on this formal model, but many practical implementations allow for some of the formal requirements to be relaxed. For example it is common to allow continuous state spaces (see section 4.6) and some implementations (Gullapalli, 1992) even allow for continuous action spaces. Many implementations will also work fine with a non-stationary environment, as long as the environment changes slowly.

4.1.1 The Markov Property

The formal reinforcement learning model can be formalized even further by introducing the Markov property. When an agent is at time-step \( t \), the agent is given the information about state \( s_t \), and it must use the information contained in this state to predict the outcome of taking different actions. If the outcome of taking an action \( a_t \) is only dependent on the current state \( s_t \) and not dependent on any of the prior states \( s_{t-1}, s_{t-2}, \ldots, s_0 \), any of the prior actions \( a_{t-1}, a_{t-2}, \ldots, a_0 \) or any of the prior rewards \( r_{t-1}, r_{t-2}, \ldots, r_1 \), the state has the Markov property and it is a Markov state. If all the states in the environment has this property, the environment has the Markov property and it is a Markov environment. Formally the Markov property can be defined by the identity of two probability distributions\(^1\):

\[
\begin{align*}
\Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} &= \Pr\{s_{t+1} = s', r_{t+1} = r | s_t, s_{t-1}, \ldots, s_0, a_t, a_{t-1}, \ldots, a_0, r_{t-1}, \ldots, r_1\} \\
(4.1.2)
\end{align*}
\]

for all \( s' \in S \), \( r \in R \) and all possible values of \( s_t, s_{t-1}, \ldots, s_0, a_t, a_{t-1}, \ldots, a_0 \) and \( r_t, r_{t-1}, \ldots, r_1 \), where \( R \) is the possible values for the reward (usually the possible values for the reward is \( \mathbb{R} \), but the mathematics become simpler if \( R \) is finite).

Equation (4.1.2) states that the probability of \( s_{t+1} = s' \) and \( r_{t+1} = r \) is the same when \( s_t \) and \( a_t \) is taken into account as when all the previous states, actions and rewards are taken into account.

The Markov property is a vital basis for any reinforcement learning method, since the only information an agent is given is the current state, and it needs to learn a behavior that can choose actions on this basis. Most real-life environments will not be 100% Markov environments, but they will be approximations to a Markov environment. When an expert backgammon player plays the game of backgammon, he will not only look at the current board position, but he will also look at the earlier moves that his opponent has made. The player will use this information to figure out which strategy his opponent has, and adapt his own play accordingly. The problem of playing backgammon does not have the Markov property, since some information about the earlier states, and perhaps even earlier games can be used for the solution. The current board state is, however, a very good approximation to a Markov state and a good backgammon playing agent can be made using reinforcement learning methods (Tesauro, 1995). In practice reinforcement learning will work fine in many cases where the environment is only an approximation to a Markov environment.

\( ^1 \)Mathematically the probability that \( X \) takes the value \( Y \), when the value of \( Z \) is known, is noted as \( \Pr\{X = Y | Z\} \).
but it must be kept in mind that the agent will have problems finding a good solution to a problem if the environment is not a good approximation to a Markov environment.

Some environments will be much further from an approximation to a Markov environment, but it will often be possible to change the states so that they are a better approximation to a Markov state. If a robot is set to search for trash in a room, then it might be a good idea to include some information about where it has already been, so that it does not keep looking for trash in the same locations. By simply including this information in the state, the state has been changed from being a very poor approximation to a Markov state, to be a rather good approximation to a Markov state.

### 4.1.2 Markov Decision Processes

A reinforcement learning problem which operates in a Markov environment is called a Markov Decision Process (MDP). MDPs was first described by Bellman (1957), and have been intensively studied in the field of dynamic programming. If the state and action space is finite, the dynamics of the MDP can be described by the probability that the next state will be \( s' \), if the current state is \( s \) and the chosen action is \( a \):

\[
\mathcal{P}_{ss'}^a = \Pr\{s_{t+1} = s'|s_t = s, a_t = a\} \tag{4.1.3}
\]

and the expected value of the next reward when current state is \( s \), the next state is \( s' \) and the chosen action is \( a \):

\[
\mathcal{R}_{ss'}^a = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\} \tag{4.1.4}
\]

When \( \mathcal{P}_{ss'}^a \) and \( \mathcal{R}_{ss'}^a \) is known in advance dynamic programming can be used to find optimal polices to MDPs, as will be seen in section 4.1.4.

### 4.1.3 The Optimal Policy \( \pi^* \)

For a MDP, there exist one or more optimal policies, which are referred to as \( \pi^* \), and which has earlier been defined as a policy which maximizes the long-term (or sometimes short-term) reward.

This section will describe various measures for defining the short- and long-term reward, along with value functions that defines the cumulative reward when following a given policy. With these definitions in place the optimal policy, and methods for finding the optimal policy, can be defined.

#### The Reward Function

The short-term reward is easy to define, since it is simply the reward received in the next time-step \( r_{t+1} \), a policy which is only concerned with maximizing the short-term reward should then simply optimize the immediate reward:

\[
R_t = r_{t+1} \tag{4.1.5}
\]

If instead the policy is concerned with optimizing the long-term reward, the sum of all future rewards should be optimized:

\[
R_t = \sum_{k=0}^{T} r_{t+k+1} \tag{4.1.6}
\]

---

\(^2\)Mathematically the expected value for \( X \), when the value of \( Y \) is known, is noted as \( E(X|Y) \).
where $T$ defines some upper bound on the number of time steps. If there is no upper bound on the number of time-steps, $T$ will become $\infty$ and $R_t$ could also easily become $\infty$. This definition of $R_t$ only makes sense in periodic tasks, where you know exactly how many time steps is left of the task. If the number of time steps is not known in advance, this definition is a bit troublesome, since it may not see a reward which is $T+1$ steps in the future, and because it will get the same $R_t$ value if a goal is reached in one step, as if the goal is reached in $T$ steps.

If the policy is concerned with optimizing both for short- and long-term rewards, it is possible to do so by discounting future rewards:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

(4.1.7)

where $\gamma$ is a discount rate, in the range $0 \leq \gamma \leq 1$. When $\gamma$ is zero, the expected discounted future reward (4.1.7) is exactly the same as the short-term reward (4.1.5), and when $\gamma$ is one, it is exactly the same as the long-term reward (4.1.6), where $T$ is $\infty$.

The Value Functions

The discounted future reward is the most general $R_t$ calculation, so the optimal policy $\pi^*$, for a reinforcement learning task will be defined, as a policy $\pi$ where another policy $\pi'$ which has a higher $R_t$ (4.1.7) for any $s \in S$ does not exist. In other words, the discounted future reward by following the policy $\pi$ for all states $s \in S$ should be optimized. The discounted future reward which is obtained by following $\pi$ from a state $s$ is called the value function $V^\pi(s)$. The value function can formally be defined for MDPs as:

$$V^\pi(s) = E_{\pi} \{ R_t | s_t = s \} = E_{\pi} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\}$$

(4.1.8)

This value function is often referred to as the state-value function, because it gives the expected future reward from a specific state. Often it is also important to know what the expected return is for choosing a specific action $a$ when the agent is in some state $s$. This value function is referred to as the action-value function $Q^\pi(s, a)$. The action-value function can formally be defined as:

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

(4.1.9)

An important property of the two value functions is that they have an optimal substructure, which means that the optimal solution can be calculated local and combined to create an optimal solution for the entire environment. The optimal substructure is the basis for dynamic programming and for the action-value function it can be defined by:

$$Q^\pi(s, a) = E_{\pi} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a \right\}$$

$$= E_{\pi} \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a \right\}$$

(4.1.10)
Defining The Optimal Policy $\pi^*$

The optimal state-value function $V^*$ and the optimal action-value function $Q^*$ can be used to define an optimal policy $\pi^*$. The optimal state-value function is defined as:

$$V^*(s) = \max_{\pi} V^\pi(s)$$  \hspace{1cm} (4.1.11)

for all $s \in S$, and the optimal action-value function is defined as:

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a)$$  \hspace{1cm} (4.1.12)

for all $s \in S$ and all $a \in A(s)$. Because the optimal state-value function, must be a result of selecting an optimal action, the optimal state-value function can also be defined as a function of the action-value function:

$$V^*(s) = \max_{a \in A(s)} Q^*(s, a)$$  \hspace{1cm} (4.1.13)

Now $\pi^*(s)$ can be defined as a function of $Q^*(s, a)$, by simply saying that $\pi^*(s)$ is the action $a$ which gives the highest action-value:

$$\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s, a)$$  \hspace{1cm} (4.1.14)

This definition for $\pi^*(s)$ does in itself not have any problems, but it uses $Q^*(s, a)$ which is defined by (4.1.12), which means that the only option for calculating $\pi^*(s)$ is to try all possible values for $\pi$ which is generally not feasible. For this reason a new definition for $Q^*(s, a)$ is needed, which can be calculated more easily.

As mentioned earlier, $P^a_{ss'}$ and $R^a_{ss'}$ from equation 4.1.3 and 4.1.4 can be used to find an optimal policy, the Bellman optimality equation (Sutton and Barto, 1998) defines the optimal state-value $V^*$ by mean of these functions:

$$V^*(s) = V^\pi^*(s)$$ \hspace{1cm} using (4.1.13)

$$= \max_{a \in A(s)} Q^\pi^*(s, a)$$ \hspace{1cm} using (4.1.9)

$$= \max_{a \in A(s)} E^\pi^* \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \bigg| s_t = s, a_t = a \right\}$$ \hspace{1cm} using (4.1.10)

$$= \max_{a \in A(s)} E^\pi^* \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \bigg| s_t = s, a_t = a \right\}$$ \hspace{1cm} using (4.1.8)

$$= \max_{a \in A(s)} E \left\{ r_{t+1} + \gamma V^\pi^*(s_{t+1}) \bigg| s_t = s, a_t = a \right\}$$ \hspace{1cm} (4.1.15)

$$= \max_{a \in A(s)} \sum_{s' \in S} P^a_{ss'} \left[ R^a_{ss'} + \gamma V^\pi^*(s') \right]$$ \hspace{1cm} (4.1.16)

The step from (4.1.15) to (4.1.16) is a bit tricky, but if a closer look is taken at $P^a_{ss'}$ it can be seen that it provides a probability distribution, which can be used to calculate the expected $V^\pi^*(s_{t+1})$ as a sum of all $V^\pi^*(s')$ multiplied by their probabilities of being the next state. $P^a_{ss'}$ can also be used to calculate the expected $r_{t+1}$ using a sum of probabilities.

Similarly the Bellman optimality equation for the action-value function can be derived:
4.1.14

\[ Q^*(s, a) = E_{\pi^*} \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} \quad \text{using (4.1.10)} \]

\[ = E_{\pi^*} \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a \right\} \quad \text{using (4.1.8)} \]

\[ = E \left\{ r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \right\} \quad \text{using (4.1.13)} \]

\[ = E \left\{ r_{t+1} + \gamma \max_{a' \in A(s_{t+1})} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right\} \]

\[ = \sum_{s' \in S} \mathcal{P}^a_{ss'} \left[ \mathcal{R}^a_{ss'} + \gamma \max_{a' \in A(s')} Q^*(s', a') \right] \quad (4.1.17) \]

By applying equation (4.1.16) to (4.1.14) a definition of \( \pi^*(s) \) is defined which is a function of \( \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'} \) and \( V^* \):

\[ \pi^*(s) = \arg \max_{a \in A(s)} \sum_{s' \in S} \mathcal{P}^a_{ss'} \left[ \mathcal{R}^a_{ss'} + \gamma V^*(s') \right] \quad (4.1.18) \]

Similarly \( \pi^*(s) \) can be defined as a function of \( Q^*(s, a) \) by applying equation (4.1.17) to (4.1.14):

\[ \pi^*(s) = \arg \max_{a \in A(s)} \sum_{s' \in S} \mathcal{P}^a_{ss'} \left[ \mathcal{R}^a_{ss'} + \gamma \max_{a' \in A(s')} Q^*(s', a') \right] \quad (4.1.19) \]

4.1.4 Finding an Optimal Policy from \( \mathcal{R}^a_{ss'} \), and \( \mathcal{P}^a_{ss'} \)

The new definitions for \( \pi^*(s) \) (4.1.19 and 4.1.18) still requires that \( V^*(s) \) or \( Q^*(s, a) \) is known, which in turn means that \( \pi^*(s) \) can be calculated by trying all possible values for \( \pi \). This approach is not very practical and a number of dynamic programming algorithms have been developed, which exploits the recursive definitions of the Bellman optimality equation (4.1.16) and (4.1.17), to speed up the process when \( \mathcal{P}^a_{ss'} \) and \( \mathcal{R}^a_{ss'} \) is known. There are two conceptually different methods which are often used when finding \( \pi^* \) using dynamic programming: Value iteration and policy iteration, which will be briefly explained here.

Value Iteration

Value iteration finds the optimal deterministic policy \( \pi^*(s) \), by first calculating the optimal value function \( V^*(s) \) by using equation (4.1.16) and then finding the optimal policy \( \pi^*(s) \) by using equation (4.1.18).

Algorithm (1) displays the general value iteration algorithm, and a formal proof of its convergence is available in Littman et al. (1995).

Policy Iteration

While value iteration only updates the policy once after having found \( V^* \), policy iteration updates the policy during each iteration, and uses this modified policy in the next iteration. However, the underlying equations (4.1.16) and (4.1.18) are still the same.

Algorithm (2) displays a policy iteration algorithm. This algorithm is a modification of the original policy iteration algorithm, which is e.g. shown in Kaelbling et al.
Algorithm 1 Value iteration algorithm

for all $s \in S$ do
    Initialize $V(s)$ arbitrarily from $\mathbb{R}$
end for

repeat
    $V' \leftarrow V$
    for all $s \in S$ do
        $V(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P_{ss'}^{a} \left[ R_{ss'}^{a} + \gamma V(s') \right]$
    end for
until difference between $V'$ and $V$ is small enough

for all $s \in S$ do
    $\pi(s) \leftarrow \arg \max_{a \in A(s)} \sum_{s' \in S} P_{ss'}^{a} \left[ R_{ss'}^{a} + \gamma V(s') \right]$
end for

Algorithm 2 Policy iteration algorithm

for all $s \in S$ do
    Initialize $V(s)$ arbitrarily from $\mathbb{R}$
    Initialize $\pi(s)$ arbitrarily from $A(s)$
end for

repeat
    $\pi' \leftarrow \pi$
    repeat
        $V' \leftarrow V$
        for all $s \in S$ do
            $V(s) \leftarrow \sum_{s' \in S} P_{ss'}^{\pi(s)} \left[ R_{ss'}^{\pi(s)} + \gamma V(s') \right]$
        end for
    until difference between $V'$ and $V$ is small enough

for all $s \in S$ do
    $\pi(s) \leftarrow \arg \max_{a \in A(s)} \sum_{s' \in S} P_{ss'}^{a} \left[ R_{ss'}^{a} + \gamma V(s') \right]$
end for

until $\pi' = \pi$

(1996). The original algorithm, needs to solve a system of linear equations, to calculate $V^\pi(s)$, which can be very time consuming for large problems while this modified algorithm uses an iterative approach. Usually when reinforcement learning literature refers to policy iteration, they are actually referring to this modified algorithm. A formal proof of its convergence is available in Puterman and Brumelle (1979).

It it hard to get a clear answer to the question of which algorithm that performs best overall, but Pashenkov et al. (1996) tries to answer this question and concludes that value iteration is usually faster, but that even faster convergence can be received by combining value and policy iteration. The major obstacle for these two algorithms is, however, not the speed of the algorithms, but the fact that they require that $P_{ss'}^{a}$ and $R_{ss'}^{a}$ is known in advance, which is not the case for the majority of reinforcement learning problems.
4.2 Learning With or Without a Model

Section 4.1.4 explains how an optimal policy can be found when the probability of getting from state $s$ to state $s'$ by choosing action $a$ ($P_{ss'}^a$) is known, and the reward function $R_{ss'}^a$ is known. However, in the general reinforcement learning problem, these functions are not known in advance, and reinforcement learning algorithms are divided into two groups:

- **Model-based learning** algorithms, which try to estimate $P_{ss'}^a$ and $R_{ss'}^a$, before using a dynamic programming method to calculate $\pi^*$.
- **Model-free learning** algorithms, which does not try to estimate $P_{ss'}^a$ and $R_{ss'}^a$, but rather try to estimate $Q^*$ directly.

### 4.2.1 Model-Based Learning

A naive model-based algorithm will try to estimate $P_{ss'}^a$ and $R_{ss'}^a$ by exploring the environment, and when the estimates converge, a dynamic programming algorithm will be used to find the optimal policy. This algorithm will need to do as much exploration as possible, so it can choose a random action, choose the least tried action or it can use a more directed exploration policy as discussed in section 4.3.5. More advanced model-based learning algorithms will be discussed in section 4.7. However, for now I will state some of the obvious problems that the naive model-based approach will have:

1. While the algorithm explores the environment, it will only try to explore, and not try to exploit some of the knowledge it has gained about the optimal policy, which means that very little reward will be received while exploring. For many real-life reinforcement learning problems there is some kind of cost associated with gaining this experience (this could simply be CPU cycles, but it could also be time slots on expensive industrial robots), so it is often desirable to get a good amount of reward, even in the initial learning phase.

2. If the state-action space is very large, the exploration strategy will try to explore all of it, although it may not be possible to do so within reasonable time. The strategy will use just as much time in seemingly hopeless areas of the state-action space, as it will use in promising areas.

3. The last problem with the simple model-based approach is the value- or policy-iteration that must be executed after estimating $P_{ss'}^a$ and $R_{ss'}^a$, which can be very time consuming because it needs to explore all state-action pairs.

### 4.2.2 Model-Free Learning

A naive model-free approach could be the greedy approach, which simply tries to gain as much reward as possible, by always selecting the action that has the highest expected reward ($\arg\max_{a \in A(s)} Q(s,a)$), where $Q$ is the current best approximation to $Q^*$. The benefit of this approach is that all the knowledge that is gathered, is used immediately to gain reward quickly and very little time is spent on seemingly hopeless solutions. This on-line approach, where the knowledge that is gained by interacting with the environment is used to direct the search for an optimal policy, is referred to by Sutton and Barto (1998) as one of the key properties in any reinforcement learning algorithm, and is what distinguishes reinforcement learning from other approaches to finding optimal policies for MDPs. This pure greedy approach will, however, often lead to an agent finding a suboptimal solution, and sticking to that solution without ever finding other more optimal solutions, so the chance of actually finding $\pi^*$ is slim.
4.2.3 Model-Free versus Model-Based

These naive model-based and model-free approaches are the two extremes, where the algorithm is either fully explorative or fully exploitive. Both algorithms have a very hard time finding an optimal solution for large problems. The first because it is not directed enough, and the second because it is too directed towards local rewards.

The primary focus in this thesis is scaling reinforcement learning towards large problems by using advanced ANN training algorithms. Model-Based methods have shown very promising results, as will be discussed further in section 4.7, but they fail to meet the central criteria of being able to scale, by using function approximation. Although function approximation can be used to learn $P_{a,s}'$ and $R_{a,s}'$, the dynamic programming algorithm will still need to traverse all possible states and actions, and I have not seen any approaches that effectively scale model-based methods towards large problems. Model-free methods only need to estimate $Q^*$ in order to navigate the environment and $Q^*$ can easily be approximated using function approximation. Since an optimal policy can be obtained directly from $Q^*$ model-free methods need not traverse all states and actions to update the policy. For this reason, the primary focus in this thesis will be on model-free methods.

4.3 Exploration versus Exploitation

Both model-based and model-free methods need some kind of action-selection strategy, which ensures that there is a balance between exploration and exploitation. The problem is to find a good action-selection strategy, which uses the right amount of exploration and the right amount of exploitation. It is, however, hard to say what the right amount is, since there are some problems where a fully explorative or fully exploitive approach is actually the most optimal approach. There exists several different heuristic algorithms, which can be used to get a good balance between exploration and exploitation, and there even exists algorithms like R-Max (Brafman and Tennenholtz, 2002) that can guarantee certain bounds. I will now discuss the benefits and drawbacks of some of the most popular action-selection strategies.

4.3.1 The $\epsilon$-greedy Selection

A very simple approach to include exploration into the greedy approach is to choose the action with the highest expected reward most of the time, and select a random action the rest of the time. $\epsilon$-greedy selection takes this approach, by selecting the greedy action $1 - \epsilon$ of the time, and selecting a completely random action the rest of the time, where $0 \leq \epsilon \leq 1$.

This action-selection approach is widely used, is very easy to understand and actually performs quite well most of the time, although it does not provide reasonable worst-case bounds. It does however have three significant problems:

1. When, at some point, it has found an optimal solution, it will still choose a random action $\epsilon$ of the time, so although it knows the optimal policy, it will not follow it.

2. When a random action is chosen, there is just as much chance of selecting an action with a low $Q(s,a)$ as an action with a high $Q(s,a)$, and there is just as much chance of selecting an action that has been tried a lot of times before, as selecting an action that has never been tried before.
3. If the optimal path requires that the agent goes through an area of the state-action space with high negative reward, finding the optimal policy will require many non-greedy actions in a row. $\epsilon$-greedy selection does not encourage this behavior, and finding the optimal policy may take a very long time.

The first problem can be solved by annealing $\epsilon$ over time, but the second problem requires that the action that is selected is chosen in another way than simple random selection. The third problem will be discussed further in section 4.3.5 where directed exploration is discussed.

### 4.3.2 Boltzmann-Gibbs Selection

Instead of simply selecting a random action $\epsilon$ of the time, without looking further at the individual $Q(s, a)$ values, Boltzmann-Gibbs distribution uses $Q(s, a)$ for the individual actions to calculate a probability of selecting the actions:

$$P(s|a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A(s)} e^{Q(s,a')/\tau}}$$

where $P(s|a)$ is the probability for selecting action $a$ in state $s$ and $\tau$ is a temperature variable, $\tau > 0$. A low temperature means that the selection will be more greedy while a high temperature means that it will be more random. The major advantage of this selection method is that the individual $Q(s, a)$ is taken into account when selecting an action, meaning that more exploration will be done when the $Q(s, a)$ values are similar, and less when there is one action that has a significantly higher $Q(s, a)$ value than the others. This approach does, however, have a major disadvantage: Many reinforcement learning problems have large areas, where the $Q(s, a)$ values are similar. This observation implies that the action leading to a state that is one step closer to the goal only has a slightly higher $Q(s, a)$ value than the action leading to a state that is one step further away from the goal. This observation means that the probability of selecting the two actions will also be close to each other, although one of the actions is clearly better than the other. Kølle (2003) documents how this fact can cause the agent to make zig-zag movements towards the goal. Wiering (1999) argues that an initial bad experience with selecting a specific action may lead to a situation where the chance of selecting that action again will be very small. Annealing $\tau$ may be a way of avoiding this situation, since it will allow for more exploration in the initial phases and better exploitation in the later phases.

### 4.3.3 Max-Boltzmann Selection

The problem of getting Boltzmann-Gibbs to select the correct action when $Q(s, a)$ values are similar can be overcome by combining $\epsilon$-greedy selection with Boltzmann-Gibbs selection. In this approach the greedy action is selected $1 - \epsilon$ of the time and a selection according to $P(a|s)$ is performed the rest of the time.

This approach combines the best from $\epsilon$-greedy selection and Boltzmann-Gibbs selection and the experiments by Kølle (2003) suggests that it is a better approach than the two individual selection rules. The Max-Boltzmann selection rule does, however, still suffer from starvation of state-action pairs, based on initial experience. One way of avoiding this starvation could be to anneal $\tau$ and/or $\epsilon$, but if the variables are annealed too quickly, the problem will still be present. Another way of ensuring that some areas of the state-action space are not starved too much is to use directed exploration, as explained in section 4.3.5.
4.3.4 Optimism in the Face of Uncertainty

Optimism in the face of uncertainty is a very simple heuristic, which is to adopt very optimistic beliefs about the rewards that can be gained by taking actions that have never been taken before. This approach is widely used, because it is so simple and because it can actually be seen as a combination between a greedy policy and a more explorative policy. The policy will start out with being very explorative and end up with being very exploitive. This strategy is also very easy to combine with other strategies like e.g. Max-Boltzmann.

This approach is the basis of some of the more directed exploration strategies, that I will present in the next section. Furthermore Brafman and Tennenholtz (2002) shows that optimism in the face of uncertainty can be proven to provide guarantees on behavior; in the R-Max algorithm it is used to provide a polynomial bound on the learning time.

4.3.5 Directed Exploration

The idea behind directed exploration is very simple: Instead of maximizing a reward value function, an experience value function is maximized. The goal of the directed exploration approach is to explore as much of the state-action space as possible before it switches strategy and starts exploiting its knowledge. The simplest directed exploration techniques are greedy techniques that, in each state, greedy selects the action with the highest expected experience value. The immediate experience value gained for selecting action $a$ when in state $s$ is $R^E(s,a)$ and the expected discounted future experience is $Q^E(s,a)$. Thrun (1999), Kølle (2003) and Wiering (1999) explore several different exploration techniques. Four basic techniques will be explained here: Frequency-based, recency-based, error-based and R-Max exploration.

Frequency-Based Exploration

Frequency-based exploration will try to explore all state-action pairs uniformly by having an experience value function of:

$$R^E(s,a) = -\frac{C(s,a)}{K} \quad (4.3.2)$$

where $C(s,a)$ is a local counter, counting the times that the action $a$ has been selected in state $s$, and $K$ is a scaling constant. This exploration strategy will always try to explore the state-action pair that has been visited least frequently. The experience value is always negative, which means that a state-action pair that has never been visited will always have a higher $R^E(s,a)$ than all the previously visited state-action pairs. Frequency-based exploration has the advantage, that it will always select the action that has been explored least frequently. This ensures that the state-action space is explored uniformly.

Recency-Based Exploration

Recency-based exploration will try to explore state-action pairs that it has not visited for a long time by having an experience value function of:

$$R^E(s,a) = -\frac{t(s,a)}{K} \quad (4.3.3)$$

where $t(s,a)$ is the last time-step where the state-action pair $(s,a)$ was visited and $K$ is a scaling constant. Both frequency- and recency-based exploration can be seen...
throughout the literature with different functions for the same goal, when recency-based exploration was introduced by Sutton (1990) he e.g. used a value based on the square root since the last visit to the state-action pair. The same approach is copied in a more advanced function by Pchelkin (2003), which incorporates the total number of states. Wiering (1999) and Kølle (2003) define a recency-based exploration value function, which is based on the current time-step $t$ instead of $t(s, a)$. This definition requires that $R^{Exp}(s, a)$ is only updated when a state is visited, since $R^{Exp}(s, a)$ would otherwise be the same for all state-action pairs.

I think that this definition can lead to confusion, and although it is common in literature, I prefer the more direct definition using $t(s, a)$.

Error-Based Exploration

Error-based exploration tries to explore areas of the state-action space, where the estimate for $Q(s, a)$ increases strongly, since it could be an area where the agent could gain positive knowledge of how to improve the policy. The experience value function is calculated by:

$$R^{Exp}(s, a) = Q_{t+1}(s_t, s_t) - Q_t(s_t, a_t) - K.$$  \hfill (4.3.4)

If the function had been changed to include the absolute difference, and not just the increase of $Q(s, a)$, then this function would simply try to explore areas where too little information about the area exists, or where the $Q$ values fluctuate a lot. But this exploration function differs, because it will only try to explore areas that seem promising. For small problems with hidden rewards as the ones explored by Kølle (2003) and Wiering (1999) this has a disadvantage because the strategy will have a hard time finding the hidden goal. For larger problems, it might however be an advantage because it will only try to explore areas that seem to give a better policy. Here, I must explain how I define small and large problems, since the problem that I refer to as small, is referred to as large by Kølle (2003). I define a large problem as any problem where good estimates of $P_{a's}$ and $R_{a's}$ cannot be found within a reasonable amount of time. The problem that Kølle (2003) defines as large is completely stationary and deterministic and has only 80,000 state-action pairs, which means that after all state-action pairs have been visited once, the estimates for $P_{a's}$ and $R_{a's}$ will be exact. I believe that this observation is the reason that he experiences such good results for the frequency-based exploration strategy, since it tries to visit as many different state-action pairs as possible, before it starts to exploit its knowledge.

R-Max Exploration

R-Max (Brafman and Tennenholtz, 2002) exploration formalizes the notation of optimism in the face of uncertainty, and states that all unvisited state-action pairs will give the same reward as the maximum reward state, and that they all lead to a goal state. R-Max uses estimates of $P_{a's}$ and $R_{a's}$, which are initialized on this optimistic basis, to compute an exploration policy which optimizes the $T$-step return. The basic idea behind this approach is actually quite simple, and can be formulated as: Either the $T$-step policy will yield high reward, or it will yield high learning. It is not possible to say which one will be the case, since unvisited states will have the same characteristics as the optimal goal state, but since there is only a polynomial number of parameters to learn, the algorithm will only use a polynomial number of explorative steps, while the rest will be exploitive. This polynomial bound is formally proven by Brafman and Tennenholtz (2002), and this is a property which makes this exploration policy very appealing.
4.3.6 Combining Selection Strategies

Purely explorative techniques and purely exploitive techniques have given good results for some kinds of reinforcement learning problems, but for the larger part some kind of combination must be used. Not that much work has been done in the field of combination techniques, but Thrun (1999) and Wiering (1999) have done some work in the field of model-free methods, while R-Max is an example of a model-based combination. R-Max is based on the work of Kearns and Singh (1998), which provides another model-based combinational approach called Explicit Explore or Exploit (E$^3$). I will, however, mostly be concerned with the model-free approaches.

Thrun (1999) describes how you statically or dynamically could combine explorative and exploitive approaches. I will not repeat his findings, but only note that it seems that a dynamic combination seems to be the best choice. I will, however, try to reason about which combinations that would make the most sense and in which situations.

If you have small problems, a very simply combination, which explores in the beginning and exploits towards the end, will be beneficial. For larger problems, this idea may not be good, since you may have a very hard time getting to the state-action pairs that have a high reward, unless you follow some kind of exploitive approach. If you e.g. learn to play a game, then you will have a very hard time winning by only following a explorative approach, so the knowledge gained from the few winnings that you do get, will not be enough to produce a good policy. If you instead follow a exploitive approach like e.g. Max-Boltzmann you should start to win more games, and the chance of producing a good policy will be higher.

For other problems where the goal is hidden behind a large negative reward, the exploitive approaches will have a very hard time finding the goal, because it will have to take a lot of non-exploitive steps before it will get to the reward. Let us imagine a very simple one-dimensional grid-world problem, with two possible actions (left and right), where there is a small positive reward at $s_{10}$ ten steps to the left of the starting position $s_0$, a large positive reward $s_{20}$ ten steps to the right of the starting position, and that the path to the high reward is filled with small negative rewards. This simple grid-world problem is displayed in figure 4.2.

In this problem, the agent will have to take ten non-exploitive steps to get to the high reward. It will take a long time before the high reward is found if a $\epsilon$-greedy or a Boltzmann approach is used, while any of the exploration-based approaches will find the large reward quickly.

[Figure 4.2: Simple one-dimensional grid-world problem which is difficult to solve for $\epsilon$-greedy and Max-Boltzmann]

It is hard to define a combinations of approaches that will work for all kinds of reinforcement learning problems, although the R-Max algorithm provides worst case guarantees, but I will try to define a combination that I believe will be beneficial for many different problems and that I believe will be able to scale to larger problem sizes. This strategy is purely speculative, but I will try to give good arguments for the individual choices.

In the beginning of the learning phase, the agent does not know much about the environment, so a strategy that uses a frequency-based exploration strategy most of the time will be highly beneficial. Later in the learning phase, the agent needs to exploit some of the knowledge that it has gained. It can do so by gradually switching
4.4 Temporal Difference Learning

Section 4.3 describes the strategies which can be used to explore the environment, but it does not describe exactly how the knowledge gathered through this exploration is used. **Temporal difference learning** (TD) methods describe how the gathered knowledge is used in a model-free way to form an optimal policy for the discounted future reward defined in section 4.1.3, but the same methods could just as
well be used to find an optimal policy for the \textit{discounted future experience} defined in section 4.3.5.

Temporal Difference learning is split into two different problems: \textit{prediction} and \textit{control}. The prediction problem is the problem of predicting the discounted future reward of following policy $\pi$ from state $s$, which means that it is the problem of predicting the state-value function $V^\pi(s)$. The control problem is the problem of learning the discounted future reward for selecting action $a$ from state $s$ and following policy $\pi$ from there, which means that it is the problem of finding the action-value function $Q^\pi(s,a)$. Section 4.4.1 will concentrate on the \textit{prediction problem}, while section 4.4.2 to 4.4.5 will concentrate on various ways to solve the \textit{control problem}.

4.4.1 Temporal Difference Prediction

The simplest form of temporal difference prediction, is the TD($0$) algorithm (Sutton, 1988), which when located in state $s$, takes the action $a$ which is given by $\pi(s)$ and observes the reward $r$ and the next state $s'$. This information is used to update the present approximation for $V^\pi(s)$ by means of the reward $r$ and the estimate for $V^\pi(s')$. If the next state $s'$ would always be the same, then the update of $V^\pi(s)$ could simply be done by discounting the future reward:

$$V^\pi(s) = r + \gamma V^\pi(s') \quad (4.4.1)$$

But since this is not the case, the prediction needs to consider the current prediction for $V^\pi(s)$ when updating it. A simple way of doing this, is by introducing a learning rate $\alpha$, where the current prediction contributes by $(1 - \alpha)$, and the new prediction contributes with $\alpha$:

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha(r + \gamma V^\pi(s')) \quad (4.4.2)$$

Another way of approximating $V^\pi(s)$ which can be used for episodic tasks is \textit{Monte Carlo prediction}. Monte Carlo prediction waits until the end of an episode and then updates $V^\pi(s)$ for all the states that have been visited, by simply looking at the actual reward which was received in all the states following $s$. Monte Carlo prediction does not use discounting in general, since the actual length of the episode is known when updating $V^\pi(s)$. The primary advantage of Monte Carlo prediction is that $V^\pi(s)$ is updated when the actual reward is received, and that all actions which lead to the reward is updated when the reward is received. There are however two disadvantages:

- Since $V^\pi(s)$ is first updated at the end of an episode, the knowledge gained during the episode is not used very efficiently. In the extreme case this could mean, that an agent could go in circles without ever finding a goal, and without being able to change its policy.
- Information about the optimal substructure is not used, which means that if one episode has been observed where the agent went from state $s$ to state $s'$ and got a reward of 1, and one episode where the agent went from state $s''$ to state $s'$ and got a reward of 0, then the estimate for $V^\pi(s)$ would be 1 and the estimate for $V^\pi(s'')$ would be 0, although the Markov property would dictate that both estimates should be 0.5.

These two disadvantages are some of the reasons that Monte Carlo methods have not received as good results as TD methods (Sutton and Barto, 1998).

Knowing $V^\pi(s)$ is in itself of little value, when nothing is known about the effect of selecting a given action ($P_{sa}$). For many problems like \textit{e.g.} board games,
some knowledge is known about $P_{as}^{a}$, since it is possible to say which state that the board will be in immediately after taking an action, but it is not known what the state will be after the opponent has made his move. This state which occurs immediately after taking an action is known as an after-state. After-states can be used directly in control, since a greedy approach could simply choose the action which leads to the after-state with the highest $V^{\pi}(s')$. This also has the advantage that two actions which lead to the same after-state are considered equal. For other problems, where no information about $P_{as}^{a}$ exists, $P_{as}^{a}$ can be learned through a separate learning function, but a more popular approach to temporal difference control is to learn $Q^{\pi}(s,a)$ directly.

4.4.2 Off-Policy $Q$-Learning

Equation (4.1.14) on page 48 states that the optimal policy $\pi^{*}(s)$ can be found if the optimal action-value $Q^{*}(s,a)$ has been found. The most popular TD control algorithms approximates an optimal policy by approximating an optimal action-value.

Section 4.3 stated that in order to gain knowledge about an environment, it is important to do some amount of exploration. Reinforcement learning algorithms are divided into two groups, the ones which estimates $\pi^{*}$ while following $\pi$ (off-policy learning), and the ones which estimates $\pi$ while following $\pi$ (on-policy learning). This basically means that on-policy methods also take the cost of exploration into account when updating $Q$, while off-policy does not take this cost into account. Section 4.4.4 will go further into advantages and disadvantages of the two, while this section and section 4.4.3 will concentrate on describing an off-policy and an on-policy learning algorithm.

The off-policy algorithm $Q$-learning updates the $Q(s,a)$ value in much the same way as TD(0) updates $V(s)$, by using a learning rate $\alpha$. When the agent is in state $s$ it takes an action $a$ based on policy $\pi(s)$ and observes the reward $r$ and the next state $s'$. Since $Q$-learning is an off-policy method it uses an approximation of the optimal $Q(s',a')$ to update $Q(s,a)$. The simplest form of $Q$-learning, which is also known as one-step $Q$-learning is defined by the update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left( r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a) \right) \tag{4.4.3}$$

The algorithm for $Q$-learning, which can be seen in algorithm (3) follows a procedure which (Sutton and Barto, 1998) refers to a Generalized Policy Iteration (GPI). This procedure is a generalization of the policy iteration algorithm from section 4.1.4, which refers to the general idea of evaluating and improving the policy iterative. The algorithm is a simple iterative algorithm, which in each iteration selects an action based on the calculated $Q(s,a)$ and a selection strategy (e.g. Max-Boltzmann), and then applies equation (4.4.3) to the result. A important aspect of the algorithm, is the fact that it does not follow the same policy as it actually updates. It follows a policy $\pi$ based on a selection strategy, but it updates the current estimate of the optimal policy $\pi^{*}$ by means of the greedy selection ($\max_{a' \in A(s')} Q(s',a')$). The advantage of this off-policy approach, is that the cost of exploration is not included in $Q(s,a)$, which makes good sense, since the policy that the algorithm finds should be executed without exploration when the learning phase is ended.

4.4.3 On-Policy SARSA-Learning

While the off-policy $Q$-learning algorithm updates the estimate for the optimal policy $\pi^{*}$, while following some explorative policy $\pi$, the on-policy SARSA-learning
Algorithm 3 One step Q-learning

for all \( s \in S \) and all \( a \in A(s) \) do
    Initialize \( Q(s, a) \) arbitrarily from \( \mathbb{R} \)
end for

for all episodes do
    \( s \leftarrow \text{some start state} \)
    while \( s \) is not a terminal state do
        \( a \leftarrow \pi(s) \), where \( \pi \) is a policy based on \( Q \) (e.g. Max-Boltzmann)
        Take action \( a \) and observe reward \( r \) and next state \( s' \)
        \[
        Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a) \right)
        \]
        \( s \leftarrow s' \)
    end while
end for

Algorithm 4 One step SARSA-learning

for all \( s \in S \) and all \( a \in A(s) \) do
    Initialize \( Q(s, a) \) arbitrarily from \( \mathbb{R} \)
end for

for all episodes do
    \( s \leftarrow \text{some start state} \)
    \( a \leftarrow \pi(s) \), where \( \pi \) is a policy based on \( Q \) (e.g. Max-Boltzmann)
    while \( s \) is not a terminal state do
        Take action \( a \) and observe reward \( r \) and next state \( s' \)
        \[
        a' \leftarrow \pi(s')
        Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma Q(s', a') - Q(s, a) \right)
        \]
        \( s \leftarrow s' \)
        \( a \leftarrow a' \)
    end while
end for

algorithm updates the same policy \( \pi \) as it follows. Instead of greedy selecting the best possible future \( Q(s', a') \), the policy \( \pi \) selects the action \( a' \) and uses this action both for updating \( Q(s, a) \) and for taking the next step. This defines the SARSA update function as:

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma Q(s', a') - Q(s, a) \right)
\]  \hspace{1cm} (4.4.4)

where \( s' \) is the next state and \( a' \) is the next action. The SARSA algorithm which is shown in algorithm (4) is very similar to the Q-learning algorithm, except for the important fact that the on-policy update rule updates \( Q(s, a) \) based on the action which is actually being taken in the next step. The name of the SARSA algorithm comes from the fact that the tuple \( < s, a, r, s', a' > \) is needed to update the action-value. While the off-line Q-learning has the advantage that it does not include the cost of exploration, the on-policy method has the advantage that because it optimizes the same policy that it follows, then the policy that it follows will be more optimal, and learning will be faster. The disadvantage is, however, that the non-explorative policy used after the learning phase is not the same as the policy which is optimized during learning.
4.4.4 Off-Policy versus On-Policy Learning

The main difference between off-policy and on-policy algorithms, is that on-policy algorithms include the cost of exploration in the approximation for $Q(s,a)$, while off-policy algorithms does not include this cost. The question about which of these approaches that is best is not simple, but I will try to explain the advantages and disadvantages for both approaches, by introducing the cliff walking problem from Sutton and Barto (1998).

The cliff walking problem, which is illustrated in figure 4.3, is the problem of getting from a start state $S$ to a goal state $G$, where the optimal route requires the agent to walk ten steps next to a cliff. If the agent falls off the cliff, the episode will stop and it will receive a high negative reward. If an agent uses an $\epsilon$-greedy policy, it will sometimes fall off the cliff, even though it has calculated the optimal $Q^*(s,a)$.

![Figure 4.3: Cliff walking problem where S is the start state and G is the positive reward goal state. Falling off the cliff will end the episode and give a large negative reward.](image)

Sutton and Barto (1998) observed that for a cliff of length 10, the $Q$-learning agent actually found the optimal policy, but because it took a random action $\epsilon$ of the time, it fell off the cliff in some of the episodes. The SARSA-learning agent on the other hand took the exploration cost into account, and chose a route which was further away from the cliff. This meant that although the SARSA-learning agent did not find the optimal route, it learned to avoid the cliff and it performed better than the $Q$-learning agent during learning. Had the $\epsilon$ value been annealed, then both the SARSA and the $Q$-learning algorithm would have found the optimal policy.

At first point, it seems that the cliff walking problem illustrates that the $Q$-learning algorithm will find an optimal policy, where SARSA-learning will not. This is not entirely untrue, but I believe that the fact that the SARSA agent learned to avoid the cliff even when it was learning is an even more interesting fact. As problems get larger, being able to navigate gets more difficult. If the cliff is 10,000 steps long, instead of 10, a $Q$-learning agent will never learn that it is dangerous to walk close to the cliff, so the agent will constantly fall off the cliff, and learning will be very slow. The SARSA-learning agent on the other hand, will learn to avoid the cliff, and will therefore learn a good policy much quicker. I believe that this is a sign that SARSA will learn faster for large complex problems while $Q$-learning will learn faster for smaller simpler problems.

Kamenetsky (2005) explores both SARSA and $Q$-learning for the game of othello, which is a large complex problem. For this problem SARSA learns faster than $Q$-learning but the learning curve for SARSA is more unstable. The end results are very similar for the two algorithms, which indicates that although the two algorithms learn in different ways, then it is hard to say which is better. An important reason to why $Q$-learning learns slower, is that it does not learn the same policy as it follows. This means that although the learning improves the approximation of the optimal policy, it does not necessarily improve the policy which is actually being
followed. An important reason to why the learning curve for SARSA is unstable, is that the explorative actions is incorporated into the observed future reward, which can lead to high variance for the reward and in turn also a high variance for the $Q$ function.

This discussion has showed the need for annealing $\epsilon$ during training. This is especially important for SARSA-learning, since the cost of taking a bad explorative action will be incorporated into the $Q$ function, which will cause the $Q(s, a)$ values to fluctuate. It is an open question, how SARSA-learning will cooperate with directed exploration methods, since they could cause the $Q(s, a)$ values to be very far away from the optimal $Q^*(s, a)$ values. I would guess, that both $Q$-learning and SARSA-learning would perform well, as long as the exploration is annealed, but I do not know of any experiments where directed exploration techniques have been used to compare SARSA and $Q$-learning for large scale problems.

4.4.5 $Q$-SARSA Learning

While SARSA and $Q$-learning techniques are similar, they still have different advantages and disadvantages. An apparent improvement would be to combine the two to form a $Q$-SARSA algorithm. The combination could use a simple linear weight parameter $\sigma$ where $(0 \leq \sigma \leq 1)$ and a $\sigma$ value of 0 would mean that the update rule would be completely $Q$-learning, and a $\sigma$ value of 1 would mean that the update rule would be completely SARSA learning. Update rule (4.4.3) and (4.4.4) can be combined to form the $Q$-SARSA update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma \left( (1 - \sigma) \max_{a'' \in A(s')} Q(s', a'') + \sigma Q(s', a') \right) - Q(s, a) \right).$$  (4.4.5)

where $s'$ is the next state and $a'$ is the next action. The $Q$-SARSA rule is novel\textsuperscript{3}, but it is not the only update rule which lies somewhere between $Q$-learning and SARSA learning. John (1994) proposed an on-policy modification to the $Q$-learning update rule, which can also be seen as a combination between $Q$-learning and SARSA learning. His modified update rule uses a weighted average over all possible actions, as seen in equation (4.4.6), instead of the simple greedy approach used by standard $Q$-learning update rule in equation (4.4.3):

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( r + \gamma \sum_{a' \in A(s')} P(s'|a')Q(s', a') - Q(s, a) \right).$$  (4.4.6)

This weighted average can be seen as a way of getting $Q$-learning to include exploration penalty in its update rule, without including the fluctuation of the SARSA algorithm. I do, however, believe that the $Q$-SARSA update rule is a more general rule, which can more easily be parameterized to include the right amount of exploration penalty.

I believe that setting the $\sigma$ parameter appropriately will enable the $Q$-SARSA algorithm to converge faster than the $Q$-learning algorithm usually does, while still maintaining a more stable learning curve than the SARSA algorithm usually does. I do not believe that the $Q$-SARSA algorithm will be able to perform substantially

\textsuperscript{3}I do not know of anyone who has suggested this combination before, and through personal communication with Michael L. Littman and Richard S. Sutton, who are two prominent reinforcement learning researchers, I have learned that they have not heard of anyone suggesting this combination either.
4.5 Eligibility Traces

When temporal difference learning algorithms like one step Q-SARSA observes a high or low reward, the only state-action pair which has its \( Q(s, a) \) value updated is the state-action pair \((s, a)\) which leads to the reward. The next time the algorithm reaches state \( s \), the state-action pair which leads to state \( s \) will have its \( Q \) value updated. This means that the reward will only be propagated one step backwards each time a state on the path is visited (hence the name one-step Q-SARSA).

If our attention is returned to the simple one-dimensional grid-world problem from figure 4.2 on page 56, it can be seen that the first time the high positive reward goal state \( s20 \) is visited, then the \( Q \) value which is updated is \( Q(s19, \text{right}) \), at some later point when \( s19 \) is visited, \( Q(s18, \text{right}) \) is updated to receive some of the reward from the goal state and so forth until finally \( Q(s10, \text{right}) \) is updated. For this simple problem, the one-step algorithm will need to take many non-exploitive actions before the optimal policy is learned. The Monte Carlo algorithms on the other hand will learn an optimal policy very quickly, because it updates the entire
CHAPTER 4. REINFORCEMENT LEARNING

path from the start state to the goal state, when the goal state is reached.

In section 4.4.1 it was established that temporal difference algorithms are usually better than Monte Carlo algorithms, but clearly TD algorithms can learn something from Monte Carlo algorithms.

4.5.1 n-step Return

A very simple way of moving one-step algorithms closer to Monte Carlo algorithms is by changing the update rule, so that it does not only update the $Q$ value for the last state-action pair, but it updates the last $n$ state-action pairs. If $n$ is two, the one-dimensional grid-world problem will update both $Q(s_{19}, right)$ and $Q(s_{18}, right)$ when $s_{20}$ is reached the first time. If $n$ is increased, to be larger than the maximum episode length, the $n$-step algorithm effectively becomes a Monte Carlo algorithm since the $Q$ values are not updated until the end of the episode.

The $n$-step return algorithm is a good way of combining Monte Carlo algorithms and an algorithm like e.g. Q-SARSA, but the simple $n$ parameter is a bit crude, and Sutton and Barto (1998) introduces the concept of averaging over several different $n$-step returns. If e.g. one-step return and 10-step return is averaged, the most recent state-action pair will get a higher return than when only using 10-step return. This approach is often beneficial since the most recently visited state-action pair is often more responsible for the observed reward than the earlier steps. Sutton and Barto (1998) proposes that the averaging is not only done over two $n$-step return functions, but over all $n$-step return functions, where one-step return is given the largest weight, two-step the second largest and so forth. This approach is known as $\lambda$-return.

4.5.2 $\lambda$-return

The idea of $\lambda$-return is that the one-step return has a weight of $(1 - \lambda)$ and leaves a weight of $\lambda$ for the rest of the $n$-step returns, iteratively this means that the two-step return has a weight of $(1 - \lambda)$ of the $\lambda$ left by the one-step return and so forth, which implies that the $n$-step return has a weight of $(1 - \lambda)\lambda^{n-1}$. The $\lambda$-return for time-step $t$ can be defined by equation (4.5.1):

$$R_{t}^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_{t}^{(n)}$$  \hspace{1cm} (4.5.1)

where $R_{t}^{(n)}$ is the $n$-step return. The sum of all the weights will always be one, which will mean that $R_{t}^{\lambda}$ is a weighted average. This definition of $\lambda$-return is very graceful, but unfortunately it is not possible to calculate it directly at time-step $t$, since calculating $R_{t}^{(n)}$ requires knowledge of the reward gained in the next $n$ time-steps, and since $n$ grows towards infinity, updates will only be possible at the end of the episode, or never for non-episodic problems.

4.5.3 Eligibility Traces

Instead of trying to calculate $R_{t}^{\lambda}$ at time $t$, $R_{t}^{\lambda}$ can continuously be approximated each time a step is taken. In order to do so, an eligibility trace $e_t(s, a) \in \mathbb{R}^+$ need to be maintained for each state-action pair. The eligibility trace $e_t(s, a)$ of a state $s$ and action $a$ is an indication of how much the current reward should influence $Q(s, a)$ at time-step $t$, or one could say that it is an indication of how much the state-action pair $(s, a)$ is eligible for undergoing learning changes (Sutton and Barto, 1998). Since more recently visited states should be influenced more by the current
4.5. ELIGIBILITY TRACES

reward, the eligibility trace of a state-action pair is decayed by $\gamma \lambda$ in each time-step, where $\gamma$ is the discount rate and the eligibility trace for the state-action pair currently visited is incremented by one. If the state-action pair at time-step $t$ is $(s_t, a_t)$, the eligibility trace of all state-action pairs at time-step $t$ can be defined as:

$$e_t(s, a) = \begin{cases} 
\gamma \lambda e_{t-1}(s, a) + 1 & \text{if } (s, a) = (s_t, a_t) \\
\gamma \lambda e_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) 
\end{cases} \forall s \in S, \forall a \in A(s) \quad (4.5.2)$$

This equation only looks back in time, as oppose to equation (4.5.1) which looks forward in time. This makes it much easier to implement this equation in a reinforcement learning algorithm like $Q$-SARSA. Sutton and Barto (1998) has detailed a proof that eligibility traces actually calculates the $\lambda$-return in the off-line case, and that it is a close approximation in the on-line case. The off-line case is the case where the updates are only performed at the end of an episode, and the on-line case is the case where the update is done at each time-step. Generally any off-line updates would like to be avoided, since learning is then prevented during an episode, and since off-line methods does not work for non-episodic tasks. The updates which are made during the episode, are what prevents the on-line methods of being an exact calculation of the $\lambda$-return, but because of the benefits of on-line updates, it actually performs better for many problems (Sutton and Barto, 1998).

Equation 4.5.2 has some minor problems, which can be overcome by a few modifications. The first problem is that it requires that all state-action pairs are updated during each episode. This problem can easily be overcome for episodic problems, since the eligibility trace of all state-action pairs, which have not been visited during the episode, will be zero, and they need not be updated. For non-episodic problems, it can be chosen to keep a complete list of all previously visited state-action pairs, but this list will quickly become very large and the time and memory required to maintain it will be a major problem. This problem can be overcome, by simply stating that all eligibility traces less than some threshold is zero, and they can then be removed from the list. In practical situations this will not cause problems, and as long as both $\gamma$ and $\lambda$ are not close to 1, the list of previously visited state-action pairs with a non-zero eligibility trace will be tolerable.

The second problem with equation (4.5.2) is that if a state-action pair $(s, a)$ is visited more than once during the episode, $e_t(s, a)$ can become larger than one. This is in itself totally correct and it is a good approximation to $\lambda$-return, but it does give the impression that taking action $a$ in state $s$ is more profitable than it might actually be. The first visit to $(s, a)$ will have a relatively low eligibility trace, and the second visit will have a relative high eligibility trace, since the first visit is longer away from the goal than the second. Equation 4.5.2 states that the actual eligibility trace for $(s, a)$ will be the sum of the two, but since no single visit to $(s, a)$ has produced so high a eligibility trace, the summation may lead to a unrealistic high $Q(s, a)$ estimate. Singh and Sutton (1996) introduces the notation of replacing eligibility traces, where the eligibility trace for the state visited is not incremented by one, but simply set to one. This eliminates the problem of too high eligibility traces, and Singh and Sutton (1996) gives both theoretical and empirical evidence that replacing eligibility traces gives better results than conventional eligibility traces. Singh and Sutton (1996) have further developed the replacing eligibility trace method, so that the eligibility trace $e(s, a)$ is set to zero for all actions $a$ that is not taken in state $s$. The reason for this approach is that if the environment is a Markov environment, then only the last taken action in any given state can be responsible for any future received reward. Using replacing eligibility traces, the equation for $e_t(s, a)$ is transformed to equation 4.5.3:
\[ e_t(s, a) = \begin{cases} 
1 & \text{if } (s, a) = (s_t, a_t) \\
0 & \text{if } s = s_t \text{ and } a \neq a_t \\
\gamma \lambda e_{t-1}(s, a) & \text{if } s \neq s_t 
\end{cases} \quad (4.5.3) \]

### 4.5.4 The Q-SARSA(\(\lambda\)) Algorithm

Eligibility traces can be used in combination with all kinds of temporal difference learning algorithms. The TD prediction algorithm with eligibility traces is called the TD(\(\lambda\)) algorithm, and similarly there exists Q(\(\lambda\)) and SARSA(\(\lambda\)) algorithms. This section will describe how the on-line version of replacing eligibility traces is implemented in the Q(\(\lambda\)) and SARSA(\(\lambda\)) algorithms, and use this as a basis for discussing how replacing eligibility traces should be implemented in the Q-SARSA(\(\lambda\)) algorithm.

#### SARSA(\(\lambda\)) Learning

In order for replacing eligibility traces to be implemented into the SARSA algorithm, the SARSA update rule from equation (4.4.4) on page 60 needs to take the eligibility trace into account, and the SARSA algorithm in algorithm (4) needs to update the eligibility trace and the \(Q\) values for all previously visited state-action pairs in each time-step.

At time-step \(t\), the SARSA update rule which needs to be executed for all previously visited state-action pairs is as displayed in equation (4.5.4). This equation uses the time-step notation \(s_t\) and \(s_{t+1}\) instead of the simple \(s\) and \(s'\) used in equation (4.4.4), because this more clearly shows which \(Q\) values are used where.

\[ Q_{t+1}(s, a) \leftarrow Q_t(s, a) + e_t(s, a) \alpha \left( r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \right) \quad (4.5.4) \]

This update rule needs to be executed for all previously visited state-action pairs \((s, a)\), but since the last portion of the equation is independent of the state-action pair which is updated, the equation can be split up into two separate equations (4.5.5) and (4.5.6):

\[ \delta_t \leftarrow \alpha \left( r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \right) \quad (4.5.5) \]

\[ Q_{t+1}(s, a) \leftarrow Q_t(s, a) + e_t(s, a)\delta_t \quad (4.5.6) \]

where equation (4.5.5) needs only be executed once during each time-step, and equation (4.5.6) needs to be executed once for each of the previously visited state-action pairs (including the current state-action pair). Inserting the two update equations, and the equation for the replacing eligibility traces (4.5.3) into the SARSA algorithm from algorithm (4) creates the SARSA(\(\lambda\)) learning algorithm.

#### Q(\(\lambda\)) Learning

Q-learning is an off-policy learning algorithm, so it is not allowed to include any exploration penalty in its \(Q\) values. This is a problem for eligibility traces, since the list of previously visited state-action pairs also include explorative steps. In order for the Q-learning algorithm to remain off-policy, it will need to clear the list of previously visited state-action pairs every time an explorative action is taken. If the policy uses very little exploration, this may not be a problem, but if much
4.5. ELIGIBILITY TRACES

exploration is used, the list will always be very short, and the effect of the eligibility traces will be negligible.

For this reason there are several different versions of the \(Q(\lambda)\) algorithm using replacing eligibility traces, the two simplest versions is the version which clears the list every time an explorative step is taken, and the version which never clears the list, just like SARSAR(\(\lambda\)). The first version is the only truly off-policy eligibility trace method for \(Q\)-learning, it was first described by Watkins (1989) and is hence referred to as Watkin’s \(Q(\lambda)\). The second version is referred to as naive \(Q(\lambda)\) by Sutton and Barto (1998), and since it does not clear the list of previously visited state-action pairs, this algorithm is a combination of on-policy and off-policy learning. Sutton and Barto (1998) also describes a third version of \(Q(\lambda)\) known as Peng’s \(Q(\lambda)\) (Peng and Williams, 1994). This version is also a combination of on-policy and off-policy learning, but it is closer to off-policy learning than the naive \(Q(\lambda)\) algorithm. Peng’s \(Q(\lambda)\) is a bit more complex than the two simple \(Q(\lambda)\) algorithms, and uses the standard one-step rule to update the \(Q\) value for the current state-action pair, but uses the greedy action to update the prior \(Q\) values.

The actual implementation of all three \(Q(\lambda)\) algorithms follows the same model as the implementation of the SARSAR(\(\lambda\)) algorithm, with the minor changes mentioned here.

\(Q\)-SARSAR(\(\lambda\)) Learning

The \(Q\)-SARSAR algorithm is a combination between an off-policy and an on-policy method, which means that both on-policy and off-policy eligibility trace methods could be used.

The pure off-policy Watkin’s method does not seem like a good choice, since only a few prior eligibility traces will be updated. Also, since the \(Q\)-SARSAR algorithm is not purely off-policy, there is no reason to try to make the eligibility traces off-policy. The more on-policy methods used by SARSAR(\(\lambda\)), naive \(Q(\lambda)\) and Peng’s \(Q(\lambda)\) seems like more feasible choices, since they use the full list of prior state-action pairs. Peng’s \(Q(\lambda)\) algorithm is altered a bit from the naive approach so that it is a bit more off-policy. I do not feel that there is any need to do this for the \(Q\)-SARSAR(\(\lambda\)) algorithm, since it is already a combination of an on-policy and an off-policy method. For this reason I will decide to use the SARSAR(\(\lambda\)) method of eligibility traces for the \(Q\)-SARSAR(\(\lambda\)) algorithm. As a future project it will however be interesting to try out different ways of calculating the eligibility traces, and see how they perform when compared to each other.

In order to change the SARSAR(\(\lambda\)) algorithm into the \(Q\)-SARSAR(\(\lambda\)) algorithm, equation (4.5.5) needs to be changed to use the \(Q\)-SARSAR update rule, as shown in equation (4.5.7):

\[
\delta_t \leftarrow \alpha \left( r_{t+1} + \gamma \left( 1 - \sigma \right) \max_{a' \in A(s_{t+1})} Q_t(s_{t+1}, a') + \sigma Q_t(s_{t+1}, a_{t+1}) \right) - Q_t(s_t, a_t)
\]

The rest of the equations from the SARSAR(\(\lambda\)) learning algorithm need not be changed, for the \(Q\)-SARSAR(\(\lambda\)) learning algorithm, so the \(Q\)-SARSAR(\(\lambda\)) learning algorithm can simply be made by combining the equations with the standard SARSAR algorithm as shown in algorithm (6).

As discussed earlier, the \(Q\)-SARSAR(\(\lambda\)) algorithm will in reality keep a list of previously visited state-action pairs, and only update \(Q\) values and eligibility traces for these state-action pairs, which greatly reduces the cost of updating these values in algorithm (6). However, the \(Q(s, a)\) values will need to be stored for all state-action pairs, which might pose a problem for large or continuous state-action
Algorithm 6 The $Q$-SARSA($\lambda$) learning algorithm with replacing eligibility traces

for all $s \in S$ and all $a \in A(s)$ do
    Initialize $Q(s, a)$ arbitrarily from $\mathbb{R}$
end for

for all episodes do
    for all $s \in S$ and all $a \in A(s)$ do
        $e(s, a) \leftarrow 0$
    end for

    $s \leftarrow$ some start state
    $a \leftarrow \pi(s)$, where $\pi$ is a policy based on $Q$ (e.g. Max-Boltzmann)

    while $s$ is not a terminal state do
        Take action $a$ and observe reward $r$ and next state $s'$
        $a' \leftarrow \pi(s')$
        $\delta \leftarrow \alpha \left( r + \gamma \left( 1 - \sigma \right) \max_{a'' \in A(s')} Q(s', a'') + \sigma Q(s', a') \right) - Q(s, a)$
        $e(s, a) \leftarrow 1$
        for all $a'' \in A(s)$ do
            $e(s, a'') \leftarrow 0$
        end for
        for all $s'' \in S$ and all $a'' \in A(s'')$ do
            $Q(s'', a'') \leftarrow Q(s'', a'') + e(s'', a'') \delta$
            $e(s'', a'') \leftarrow \gamma \lambda e(s'', a'')$
        end for
    end while

$s \leftarrow s'$
$a \leftarrow a'$
end for

spaces, and in this case some kind of function approximation will need to be used. Section 4.6 describes how function approximation can be used in combination with reinforcement-learning, in order to scale reinforcement learning towards large problem sizes.

4.6 Generalization and Function Approximation

When the state-action space becomes large or continuous two major problems arise:

1. It may not be possible to store $Q(s, a)$ values for all state-action pairs in the available memory.

2. It may not be possible to reach all state-action pairs within a reasonable amount of time.

The first problem is very tangible, but if the $Q(s, a)$ values are e.g. stored in a database, it will not really pose a problem. The second problem is more problematic. If it is not possible to reach all state-action pairs, it will be very difficult to learn anything useful. This is especially a problem if the state-space is continuous, since the chance of reaching the exact same state several times is very small. If the algorithm does not reach the same state several times, then the learned values will be of no use in traditional tabular reinforcement learning.

For this reason some kind of generalization is a necessity. Several kinds of generalization exists, but I will focus on function approximation. Function approximation
can be used to learn several different functions within the reinforcement learning domain, but it is usually used to learn either $V^\pi(s)$ or $Q^\pi(s, a)$. If these functions are learned, many of the standard reinforcement learning algorithms like TD($\lambda$), $Q(\lambda)$, SARSA($\lambda$) and $Q$-SARSA($\lambda$) will be able to work, without major modifications.

When function approximation is used in the $Q$-SARSA($\lambda$) algorithm, the function approximator learns a reward $r$ from taking an action $a$ in a given state $s$ and following the policy $\pi$ from there on:

$$r = Q^\pi(s, a) \quad (4.6.1)$$

where $s$ and $a$ are input to the function approximator and $r$ is the output. This function approximator can be used to estimate the reward of taking all the possible actions in a given state and following policy $\pi$ from there on, and the result of this estimate can be used to select the action with the best reward. Another approach, which is used by Rummery and Niranjan (1994) and Vamplew and Ollington (2005) is to train a single network for each actions, thus making the actions independent of each other, but also adding to the complexity of the solution.

A problem which must be addressed when using a function approximator, is how the function approximator should be trained, and what it should optimize for. Normal tabular algorithms will simply overwrite an entry in the table, each time a new $Q(s, a)$ value is calculated. For function approximation this is not as simple, partly because it is not possible to overwrite a $Q(s, a)$ value in the function approximator, and partly because generalization is needed, which implies that simply overwriting is not the best choice. The function approximator can not give exact results for all $Q(s, a)$ values, so it must be decided which values that the function approximator should focus on, and hence which values the function approximator should be trained on. Some kind of uniform sampling in the state-action space could be used, but a method which is usually much more effective is to train with the state-action pairs which are actually visited during training, this way the function approximator uses more focus on state-action pairs which are visited often, and less focus on infrequently visited state-action pairs. Another advantage of this approach is that it is simple to implement, especially when using an on-line training method as e.g. the incremental neural network training from section 2.2.2, since then training can be done by simply training one iteration with the new $Q(s, a)$ value.

Since the function approximator should be able to generalize over the input it receives, the encoding of the states and actions is of importance. The states and actions should be encoded, so that it is easy for the function approximator to distinguish between two conceptional different state-action pairs, but still allow for generalization to happen. This implies that two state-action pairs which shares characteristics in the problem, should also have similar encodings. For a simple grid-world problem the encoding of the state could e.g. be an $x$ and an $y$ value. This encoding will be very effective if states that are close to each other in the grid also have a tendency to have $Q^\pi(s, a)$ values that are close to each other. The actions could be encoded as a single integer value, where $\{1 = \text{up}, 2 = \text{down}, 3 = \text{left}, 4 = \text{right}\}$. This encoding will however seldom be a good encoding, since it implies that the action $\text{up}$ has more in common with the action $\text{down}$, than it has with the action $\text{right}$. For this reason the recommendation in this case would be to let the four actions be represented as four distinct inputs.

Representation of inputs to function approximators is a challenging problem, and it is often hard to tell whether one representation will be better than the other, since it will often require knowledge of the problem which the function approximator is trying to solve.

When function approximation is used in combination with the $Q$-SARSA($\lambda$) algorithm, very large state-action spaces can be represented in a compact and effective
representation. Even continuous states can easily be represented, but continuous actions are a completely different matter. The algorithm will still have to go through all actions in $A(s)$, in order to find the action with the highest expected future reward. If $A(s)$ is very large, this process will be very time consuming, and if $A(s)$ is continuous it will not be possible to go through all the possible actions. There are generally two ways of solving this problem, while still using the $Q$-SARSA($\lambda$) algorithm; either the actions can be made discrete, or the selection strategy can make a search through the action space in order to find a good action. However, these two solutions will loose some precision, and no guarantees for their convergence can be given. Other more advanced algorithms have been proposed to solve the problem of continuous actions. Ravindran (1996) and Smith (2001) explain some of these algorithms.

The convergence of reinforcement learning with function approximation is not as straight forward as for the tabular case. Proofs exists that some reinforcement learning algorithms will converge under specific assumptions. Tsitsiklis and Roy (1996) proves that TD algorithms with linear function approximators converge to an approximation of the optimal policy, and Gordon (2000) proves that the SARSA algorithm with linear function approximation converges to a region. However, the same proofs do not exists when using a non-linear function approximator like e.g. a neural network. Proven convergence is an important aspect of reinforcement learning, but unfortunately the convergence proofs often require that an infinite amount of time is used, so in practical situations these proofs can only serve as guidance, and can not be used to ensure that implementations converge within a limited time. However, these theoretical proofs for the different algorithms are very powerful tools for validating empirical results. It is e.g. interesting to learn that it is harder to prove convergence for $Q$-learning with function approximation, than it is to prove the same for SARSA learning (Gordon, 2000).

4.6.1 Function Approximation and Exploration

The $\epsilon$-greedy and Max-Boltzmann selection strategies described in section 4.3 are simple to use with function approximation, although it is not simple to use them in combination with optimism in the face of uncertainty. However, the more directed approaches are not as easy to use directly with function approximation. Frequency-based exploration could use function approximation to approximate $C(s, a)$, but I do not believe that the approximation will be very accurate, because the function that should be approximated is constantly changing. The same applies to recency-based exploration where estimates for $t(s, a)$ could be made through function approximation. I do not know of anyone who have tried either of these approximations, and as mentioned, I do not believe that good approximations of these functions could be made.

Error-based exploration would be possible to do with function approximation, and I believe that the approximation could be made good enough for the algorithm to work in practice. However, I do not know of any experimental results using this algorithm with function approximation. This section will describe confidence based exploration and tabu search which are exploration techniques that can easily be combined with function approximation.

Confidence Based Exploration

Thrun (1999) experiments with a confidence based exploration strategy, where he uses a confidence on the estimates for an underlying model network, to direct the exploration. The exploration is directed to explore areas where the confidence of the model is low, in order to increase confidence. This exploration method is combined
with *selective attention*, to create a policy which sometimes explores, and sometimes exploits its gathered knowledge. The experiments with this method give very good results for the benchmark problems that *Thrun* (1999) used. *Wiering* (1999) uses an exploration policy very similar to this, as an example of a false exploration policy, since this policy will use much time exploring areas which have a very stochastic $\mathcal{P}_{aa^l}$, instead of exploring areas which have been infrequently visited. The problem that *Thrun* (1999) tried the confidence based approach on, did not suffer from this problem, because the environment had a uniform distributed stochasticity, and I believe that this is the case for many reinforcement learning problems. However, for some kinds of problems confidence based exploration may be problematic.

**Tabu Search**

The exploration/exploitation dilemma can be seen as a dilemma of exploiting a local optimum versus exploring to find a global optimum. This dilemma is not exclusive to reinforcement learning, but is common for many areas of computer science. Common to all of these areas is that some kind of function needs to be optimized, the difference is how solutions to the optimization problem is found. Often solutions that work well in one domain, will also work well in other domains, but this is not always the case. In the field of mathematical optimization, one of the more popular heuristics for finding optimal solutions is the local search algorithm known as tabu search (*Glover and Laguna*, 1993), which locally searches the optimization space, while making sure not to return to a place in the optimization space which has recently been visited. This heuristic is popular in combinatoric optimization, because it has the power of escaping a local optimum while still searching for globally optimal solutions. This approach can be seen as a directed exploration approach, because it forces the exploration of areas which have not been visited for a while, but it still tries to exploit the information that it has gathered.

For reinforcement learning the optimization space would be the space of all possible policies, so using a local search algorithm directly would generally mean that the policy should be evaluated by executing it for a few periods, where-after the policy should be changed a bit and so forth. This would be a totally different strategy than previously followed, and I do not believe that it will be a beneficial strategy. If a closer look is taken at tabu search, the principle is simply; that if the agent has just followed one policy, the agent should wait a bit before it tries that policy again. One way of implementing this in reinforcement learning would be to put some constraints on the state-action pairs which are visited. However, just because the same action $a$ is selected in the same state $s$ two times in a row, does not mean that the same policy is followed; it just means that the small part of the policy which involves $(s, a)$ has not been changed.

*Abramson and Wechsler* (2003) have successfully applied tabu search to the SARSA algorithm with function approximation, they used a list of recently visited state-action pairs as their tabu elements, and in order to make the algorithm converge to a single policy, they allowed the greedy action to be taken, if it was within a confidence interval. The confidence interval tells something about how confident the algorithm is of the estimates of the $Q$ values, so in this case the algorithm allowed greedy actions to be taken even though they would violate the tabu list, when the algorithm was very confident in its estimate of the $Q$ value.

I believe that tabu search can be a relatively simple way of adding exploration to a reinforcement learning algorithm using function approximation, but I do not necessarily agree that using a confidence interval is the best approach, especially since *Thrun* (1999) used a similar confidence to do the opposite. *Abramson and Wechsler* (2003) used the confidence to ensure convergence while *Thrun* (1999) used the confidence to ensure exploration. So perhaps other methods besides the confidence
interval could be used to ensure convergence. I could think of a few methods which I believe could be effective, without having to use a confidence interval:

- Simple annealing of the tabu list, making the list shorter during learning, until a greedy approach is followed.
- Randomly allowing the tabu list to be violated, with an annealed parameter selecting how often it can happen.
- Using another selection method like \textit{e.g.} Max-Boltzmann and only inserting the explorative actions into the tabu list, this will assure that the explorative actions that Max-Boltzmann take are not the same each time.

The last of these suggestions has been implemented for this thesis in combination with both Max-Boltzmann and \(\epsilon\)-greedy selection. None of the other explorative approaches have been implemented, but I believe that it could be interesting to do further tests with the \textit{confidence based exploration} approach by Thrun (1999).

4.7 Model-Based Learning

The primary part of this chapter has been spent on \textit{model-free} methods, that learn a \textit{value function} \(V(s)\) or an \textit{action-value function} \(Q(s, a)\) in order to solve the reinforcement learning problem. Learning these two functions is known as a \textit{model-free} way of solving the reinforcement learning problem, since the policy is learned directly, without learning a model of the environment.

\textit{Model-based} learning methods were introduced in section 4.2.1 on page 51. This section will give a broader view of the model-based approaches, and section 4.7.1 will explain how model-based methods can be combined with model-free methods. Model-based methods learns a model of the environment, by approximating \(P_{ss'}^a\) and \(R_{ss'}^a\). This model is then used to produce a policy, by means of dynamic programming \textit{e.g.} value or policy iteration. Section 4.2.3 states that a simple algorithm using this approach will have a very hard time scaling towards larger problem sizes, but model-based methods, however, have other advantages.

Model-based learning algorithms uses the information that is gathered during training very effectively. Because they try to learn a model of the environment before learning the actual policy, they are able to combine information from different experiences in a way which is usually not possible for model-free methods. Let us imagine the simple grid-world problem displayed in figure 4.4, with one start state, one positive reward goal state and one negative reward goal state. If the only experience that is gathered about the problem is the two runs displayed in figure 4.4, then a cognitive person would be able to piece the two runs together and produce the optimal path. A model-free algorithm would not be able to do so, since its experience tells it that moving along the bottom part of the grid will give a negative reward. A model-free approach is however not concerned with the individual runs, but sees the whole picture. It knows where the positive reward goal is, and it can figure out how to easiest get there, so it will be able to produce the optimal path from these two experiences.

This ability to use the gathered experience effectively is a very attractive property of model-based learning algorithms. Model-based learning has shown some really promising results in the past couple of years, but research is going in many different directions at the moment and it is hard to predict where it will end up. Some of the more advanced model-based learning algorithms have overcome some of the issues about scaling to larger problem sizes, but they still suffer from an inability to scale to very large and continuous problems. Strehl et al. (2006a) explores methods of speeding up the dynamic programming part of model-based algorithms and
4.7. MODEL-BASED LEARNING

First run  Second run  Optimal path

Figure 4.4: Two sample runs in a simple grid-world problem, and the optimal path based on the information gathered from the two first runs.

are presenting an approximation algorithm that only requires $O(\ln^2(|S|))$ computational steps for each time-step, where $|S|$ is the number of states. The results seem promising and it will be interesting to follow the development of the model-based reinforcement learning algorithms in the future.

Some of the most interesting model-based learning algorithms at the moment are R-Max (Brafman and Tennenholtz, 2002), Model-Based Interval Estimation (MBIE) (Strehl and Littman, 2004) and Model-Based Policy Gradient methods (MBPG) (Wang and Dietterich, 2003). These model-based methods are a bit more complicated than simply approximating $P_{ss'}$ and $R_{ss'}$, but I will refer to the cited literature for further explanation of the algorithms. Combining some of these model-based algorithms with function approximation is also possible, but unfortunately this approach does not eliminate the need for a dynamic programming part of the algorithm, so the time complexity is still high. As far as I know, no work has been done to try to combine cascading neural networks with model-based reinforcement learning.

Some of the model-based algorithms have been proven to produce near-optimal policies, with high probability, after a polynomial amount of experience (Strehl and Littman, 2005), which is a better convergence guarantee than the model-free methods can usually provide. The novel model-free Delayed Q-learning algorithm by Strehl et al. (2006b), does, however, provide a similar guarantee, so there is hope that other model-free approaches may also be able to provide the same guarantees. Model-based reinforcement learning is a large research area in active development, and I believe that this area of reinforcement learning will be very fruitful in the future. However, the algorithms still have problems scaling to large and continuous problems. For this reason I will primarily focus on model-free methods, but I believe that an interesting future project would be to combine cascading neural networks with model-based methods.

4.7.1 Combining Model-Based and Model-Free Learning

One of the major drawbacks of the model-based methods is that they require dynamic programming in order to derive a policy from the model. Model-free methods on the other hand does not require this, which makes it easier to scale model-free methods to large scale problems.

It could however be possible to incorporate simple models into model-free methods, hence making more effective use of the gathered knowledge. Most of the work that has been made in this area involves maintaining a modified model of $P_{ss'}$ and $R_{ss'}$. $P_{ss'}$ is modified, so that instead of learning a probability, it learns a function $P_s$ which takes a state $s$ and an action $a$ as input and delivers a state $s'$, which is the state that is most likely to follow from taking action $a$ in state $s$. $R_{ss'}$ is modified to a function $R_s$ that only uses $s$ and $a$ as input parameters, and not the
next state $s'$. Using $P^a_s$ and $R^a_s$ it is possible to update the $Q(s, a)$ for some state $s$ and action $a$, by simply executing the following equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left( R^a_s + \gamma \max_{a' \in A(P^a_s)} Q(P^a_s, a') - Q(s, a) \right)$$ (4.7.1)

This approach is used in algorithms like Dyna-Q and prioritized sweeping (Sutton and Barto, 1998). These algorithms have shown good results, but I will however propose two other ways of incorporating models in otherwise model-free algorithms. Since one of the major benefits of model-free methods is that they can scale to larger problem sizes by using function approximation, it must be a requirement for these methods, that they are able to use function approximation.

**Forward modelling**

The idea of forward modelling is that a model is used to implement a simple form of lookahead. This lookahead is only used for planning, so no Q values are learned from the model. The simple lookahead uses one-step lookahead to determine the best greedy action $a_{\text{greedy}}$ based on a model of $P^a_s$ and $R^a_s$. Instead of selecting the greedy action by $\arg \max_{a \in A(s)} Q(s, a)$, the greedy action can be selected by:

$$\arg \max_{a \in A(s)} \left( Q(s, a) + \alpha \left( R^a_s + \gamma \max_{a' \in A(P^a_s)} Q(P^a_s, a') - Q(s, a) \right) \right)$$ (4.7.2)

The lookahead can of course also be used to look more than one step into the future, but since $P^a_s$ is only an estimate, the results of the lookaheads may be too inaccurate. Another problem with lookahead too far into the future is the computational overhead, if there are many possible actions for each state, the computational overhead may even be too high for simple one-step lookahead.

Lookahead could of course be combined with some of the methods that update $Q(s, a)$ on the basis of the model. However, I do believe that some of the strength of lookahead, lies in the fact that it does not alter the $Q$ values based on a approximative model, but only selects actions based on this model.

**Backward modelling**

Backward modelling tries to do the exact opposite of forward modelling. Instead of trying to learn the next states, it tries to learn the previous state-action pairs. The function approximator needs to learn a function which takes the current state $s$ as input and delivers the most likely previous state-action pair. This function can be used to add a backward modelling eligibility trace to the reinforcement learning algorithm. Where the normal eligibility trace propagates reward down to the actually previously visited state-action pairs, this eligibility trace method will propagate reward down to the most likely previously visited state-action pairs.

Often there will be an overlap between the actual prior state-action pairs and the ones found by the function approximator, but in a situation like the simple grid-world problem from figure 4.4, this might not be the case. If the first run has been run twice, instead of only once, the eligibility trace based on the function approximator will actually propagate reward down the optimal path, since the most likely previously state-action pairs will lead down the path visited by the first run.

It is hard to say which eligibility trace method, that will be the best, but I will argue that the conventional method will usually be the best, since it is known for sure that there is a path from these state-action pairs to the current state. The backward modelling eligibility trace method does not give this guarantee, and if the
backward model is not a good approximation, it may give reward to state-action pairs, for which there does not exist a path to the current state. However, I believe that the backward modelling eligibility trace method can be used in combination with the conventional eligibility trace method, so that reward is propagated back through both paths, and I believe that this method will give better results than using the conventional eligibility trace method alone.

Another approach to backward modelling would be to simply apply equation (4.7.1) to all the state-action pairs found on the backward modelling eligibility trace.

For this thesis, no combinations of model-based and model-free methods has been implemented, since the primary focus is on improving reinforcement learning by combining it with advanced neural network training algorithms, and not on improving reinforcement learning by other means. However, I feel that it is important to include these other methods in the thesis, in order to better understand the strengths and weaknesses of the implemented methods and an interesting future project is the combination of model-based and model-free methods for large scale problems.
Chapter 5

Reinforcement Learning and Cascading ANNs

Combining reinforcement learning with the Cascade-Correlation architecture was first suggested by Tesuaro (1995), which is the same article that describes one of the most successful reinforcement learning applications TD-Gammon. Despite this, the first successful combination of the two was not published until eight years later by Rivest and Precup (2003). To the best of my knowledge, only two universities have produced results combining Cascade-Correlation and reinforcement learning. McGill University in Montreal (Rivest and Precup, 2003; Bellemare et al., 2004; Bellemare, 2006) and the University of Tasmania (Vamplev and Ollington, 2005; Kamenetsky, 2005). The two universities use different approaches to combine Cascade-Correlation and reinforcement learning, but both universities have produced very promising results for small problems and more discouraging results for larger problems. The different approaches which can be used when combining the two algorithms will be the primary focus of this chapter. I will start by explaining the existing approaches and their advantages and disadvantages in section 5.1 and 5.2, and use this discussion to propose a novel approach to combining Cascade-Correlation with reinforcement learning in section 5.4.

5.1 Batch Training With Cache

The Q-SARSA(λ) reinforcement learning algorithm is on-line and the Q(s, a) values are updated after each step, which poses a problem for any batch neural network training algorithm that should be used to approximate the Q(s, a) function, since they require that the full set of training samples is available when training occurs. Rivest and Precup (2003) and Bellemare et al. (2004) are getting around this problem, by using a variation of the mini-batch algorithm.

The mini-batch algorithm divides the complete set of training samples into minor portions (mini-batches) and then trains one epoch on each portion iteratively. This approach cannot be used directly for reinforcement learning because all training data is not available up front, and in order to use any kind of batch algorithm together with reinforcement learning, some amount of training data needs to be available before training can begin.

Rivest and Precup (2003) and Bellemare et al. (2004) use a cache, which serves two purposes; firstly it is used as a mini-batch when training the neural network, secondly it is used as a read cache, so that if a value is present in the cache, the neural network is not evaluated to get the value. The benefit of this approach is that experience gathered on-line will also be available on-line, even though the neural
network have not been trained with the data yet. Generalization will however not happen until the neural network has been trained. When the neural network has been trained, the cache is cleared, so the data in the cache will always be either a value read from the neural network, or a value which has been updated after being read from the neural network.

5.2 On-line Cascade-Correlation

The cache used by Rivest and Precup (2003) is a simple cache, which has two disadvantages; firstly the neural networks are not trained while the cache is being filled, so no generalization will happen in this period. Secondly the state-action pairs which have been visited several times are only present once in the cache, which means that they will only be used once during training, contradicting the idea that often visited state-action pairs should receive more attention than infrequently visited state-action pairs. Bellemare (2006) uses a more on-line approach, which does not have the same disadvantages. The outputs of the ANN are trained using incremental training instead of batch training, which means that generalization will happen during training, and that state-action pairs visited several times will be trained several times. During training of the output neurons, the training samples will be saved, so they can be used to train the candidate neurons. The candidate neurons are trained with batch training using the saved history of training samples. Since the candidates are trained with a simple history of previous state-action pairs, the candidates will also be trained several times with state-action pairs that are visited several times. However, it should be mentioned that if a state-action pair \((s, a)\) has been updated several times, then it will be present in the history several times, but with different \(Q(s, a)\) values. It is not obvious that this is a problem, since these are the exact values that the incremental training has also used, but the latest \(Q(s, a)\) must be the most correct, and the incremental training will also be biased towards this value. It would probably be a better idea to simply include the last \(Q(s, a)\) value several times, instead of using the experienced values for \(Q(s, a)\).

Vamplev and Ollington (2005) takes the on-line approach further and uses a fully on-line approach, which trains the outputs and the candidates in parallel using incremental training. This approach eliminates all need for a cache, and has all the advantages of on-line reinforcement learning function approximation, including being able to function in a real-time environment. It does, however, reintroduce the moving target problem which the Cascade-Correlation algorithm was designed to remove.

The two on-line approaches do not have the disadvantages that the cached version has, but they have introduced a new disadvantage, which the cached version did not have: They use incremental training for Cascade-Correlation, which is not the best choice of training algorithm for Cascade-Correlation, as described in section 3.3 on page 34. The experiments in section 3.3 can not directly be transferred to the two on-line reinforcement learning algorithms, but the experiments suggest that combining Cascade-Correlation with incremental training can be problematic.

5.3 Incremental Training versus Batch Training

When combining the batch Cascade-Correlation with the on-line reinforcement learning, there are two choices, either to make the Cascade-Correlation algorithm more on-line like suggested in section 5.2, or to make the reinforcement learning algorithm more batch like as suggested in section 5.1. In this section I will discuss some of the advantages and disadvantages of these two approaches.
5.4 A SLIDING WINDOW CACHE

Temporal difference learning is per definition on-line, which has two implications, the first being that more effort is put into frequently encountered states, and the second being that the policy is being updated after each step. The fact that the policy is being updated after each step, is one of the key advantages of temporal difference learning compared to Monte Carlo methods. The constant updating of the policy is what helps a reinforcement learning agent receive results in new or changing environments, because it will learn during an episode that e.g. walking into a wall is a bad thing. This knowledge will be used to avoid the wall and reach a goal state. If the policy is not being updated during an episode, the agent will have to rely on the stochastics of its policy and the environment to avoid the wall, which may be very difficult and for some problems it may lead to an infinite loop.

The on-line approach of temporal difference learning makes neural networks trained with incremental training an obvious choice as function approximator, since the function approximation can be implemented by simply training with the new value for \( Q(s,a) \) calculated by equation (4.4.5) in each step. Incremental training is however not always the ideal solution when training a neural network, as shown in section 3, which would imply that incremental training might not be the best solution.

Because of the obvious advantages of incremental training, very few people have tried to combine reinforcement learning with any of the batch training algorithms for neural networks, like RPROP, Quickprop and Cascade-Correlation, and I have not been able to find any literature on the subject pre-dating the results of Rivest and Precup (2003). Lin (1992) used experience replay which is in itself a batch training method, but to the best of my knowledge it was only used in combination with standard incremental back-propagation. However, the results found using experience replay suggests that training using past experiences is beneficial. This knowledge combined with the bad results for combining incremental training with Cascade-Correlation in section 3.3, argues that it will be more beneficial to use a batch Cascade-Correlation algorithm in combination with reinforcement learning, than it will be to use an on-line Cascade-Correlation algorithm. However, the batch approach used by Rivest and Precup (2003) had some disadvantages, which I will try to avoid by using the novel approach described in the next section.

5.4 A Sliding Window Cache

Instead of switching to an incremental training algorithm to avoid the problems of the cache used by Rivest and Precup (2003), another approach is to alter the cache so that it does not exhibit the same problems.

The cache used by Rivest and Precup (2003) had the problem that frequently visited state-action pairs did not appear several times in the cache, and the history window had the problem that frequently visited state-action pairs appeared with different \( Q(s,a) \) values in the window. These two problems can be eliminated by using a cache and adding a counter to the cache, so that state-action pairs that are visited several times will also be present several times in the batch training data-set. When the cache is used as a lookup cache this will also ensure that the only the latest \( Q(s,a) \) value will be returned.

The problem of generalization not happening while the cache is being filled, is a bit harder to solve, while still maintaining that the algorithm should be a batch algorithm. I propose using a sliding window cache which will function as a sliding window of the last \( n \) visited state-action pairs. After the initial \( n \) time-steps the cache will be full and batch training algorithms can be used after each time-step, which means that, after the initial \( n \) time-steps, generalization will happen for each step. In order for generalization to happen during the first \( n \) time-steps, I propose
to train the ANN using incremental back-propagation for the first $n$ time-steps. Since the sliding window cache will always include information about the last $n$ time-steps, this will mean that the mini-batch created on the basis of the sliding window cache will include one new state-action pair after each time-step, and one old state-action pair will be removed.

In order for the sliding window cache to incorporate the solutions mentioned in this section, the architecture of the cache must enable three central properties:

1. If a state-action pair $(s, a)$ is available in the cache it should be possible to make a lookup and get the latest $Q(s, a)$ value for $(s, a)$.

2. State-action pairs visited several times should be present several times in the cache, or at least have a counter telling how many times they have been visited, and they should all have the latest value for $Q(s, a)$.

3. When the cache is full and a new pair $(s_t, a_t)$ is inserted in the cache, the pair that was visited $n$ time-steps earlier $(s_{t-n}, a_{t-n})$ should be removed. If this pair has been visited since $t-n$, only the oldest occurrence should be removed.

Figure 5.1 shows an example of how the sliding window cache could look. Whenever a new state-action pair is added to the cache, it is added to the start of the queue, and the lookup table is updated. If the cache is full at this point the oldest element will be removed from the cache and this is done by removing an element from the end of the queue and updating the lookup table accordingly.

<table>
<thead>
<tr>
<th>Lookup key</th>
<th>Lookup value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(s_0, a_0)$</td>
<td>$Q(s_0, a_0)$</td>
<td>1</td>
</tr>
<tr>
<td>$(s_2, a_2)$</td>
<td>$Q(s_2, a_2)$</td>
<td>1</td>
</tr>
<tr>
<td>$(s_3, a_3)$</td>
<td>$Q(s_3, a_3)$</td>
<td>1</td>
</tr>
<tr>
<td>$(s_4, a_4)$</td>
<td>$Q(s_4, a_4)$</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5.1: Sliding window cache with a size of five, where the same state-action pair $(s_4, a_4)$ is visited in time-step 1 and in time-step 4. The Lookup key $(s, a)$ is the key used when reading $Q(s, a)$ from, or writing $Q(s, a)$ to, the cache. When a $Q(s, a)$ value is written to the cache, the Q value is updated, an element is added to the FIFO-queue and the Count is incremented. When an element is removed from the FIFO-queue the Count is decremented and if it reaches zero the element is removed from the Lookup table.

The $Q(s, a)$ values in the cache will not necessarily be the same values that are available from the ANN, because the values in the cache may have been updated since they were read from the ANN, and because the ANN may have been changed since the values were read. This was also the case for Rivest and Precup (2003), but they cleared their cache after each training session. With the sliding window cache, frequently visited state-action pairs may never leave the cache, so there may be a span between the $Q(s, a)$ value in the cache and the $Q(s, a)$ value from the ANN. This span should hopefully be small, because of the training that happens after each time-step, but there is no guarantee that it will be. With a span between the two values, the question that needs to be asked, is which of the two is the most correct value. The $Q(s, a)$ value in the cache will have the exact value calculated by the update rule, but the $Q(s, a)$ value in the ANN will have the generalization from
5.4. A SLIDING WINDOW CACHE

the other \( Q(s, a) \) values, which might actually be a more correct approximation of the real \( Q^*(s, a) \). For frequently visited state-action pairs, the calculation for \( Q(s, a) \) will be more accurate than for less frequently visited state-action pairs, so I believe that for the frequently visited state-action pairs, the most correct value is the value in the cache. For the less frequently visited state-action pairs the \( Q(s, a) \) value from the ANN is probably more correct, since they would benefit from the generalization. This would indicate, that although the cache should be kept reasonable large, it might not be a good idea to make it too large, since some of the generalization will be lost.

For many problems with a large state-action space, only a small part of the state-action space will be visited after the learning has converged. Some of these problems might benefit from the cache, since most of the state-action pairs which are actually visited will be in the cache. This should make it easier for the reinforcement learning algorithm to converge to a local optimum, since it is not as dependent on the convergence of the ANN.

Using a sliding window cache will speed up execution, since frequently visited state-action pairs will be in the cache and the ANN will not need to be executed for them. The learning will however not be as fast, since the ANN will be trained with the entire cache for each time-step. If this poses a problem, the learning could be postponed a bit, so that learning e.g. only occurs every ten time-steps. As the tests in chapter 6 shows, this approach proved to not only improve performance, but it was also effective for avoiding over-fitting of the ANN.

In the cache used by Rivest and Precup (2003), the state-action pairs that are read are inserted into the cache, and also used for training. This means that the ANN will actually be trained with values that are read from the ANN. The reason that this might not be such a bad idea, is that it helps to make sure that the ANN does not forget something that it has already learned. This process is known as rehearsing, and Rivest and Precup (2003) indicates that it is a beneficial approach. However, the later results by Bellemare et al. (2004) indicate that it might not always be beneficial. The sliding window cache could easily incorporate rehearsing, by simply including both the read and written \( Q(s, a) \) values. This means that for all visited state-action pairs, the \( Q(s, a) \) values will actually be used two times during training, since they are read before they are written, but this is actually desirable since the ANN should pay more attention to the state-action pairs that are visited.

If the state-action space is very large or continuous, the same state-action pairs will not be visited often, and the sliding window cache will more or less be reduced to a history window. The approach for training the ANN will still be the same, but the reinforcement learning will not be able to benefit as much from the lookup part of the cache. It must, however be kept in mind, that even for large discrete problems, state-action pairs close to the start state and the goal state will probably be visited several times, so they will be present in the lookup cache. Since infrequently visited state-action pairs will benefit from generalization, they will probably have a more precise value in the ANN, than in the lookup cache. With this in mind, I do not view the lack of lookup availability as a problem for the sliding window cache, but it must be kept in mind that it will not function in the same way for very large or continuous problems, as it will for smaller problems.

5.4.1 Eligibility Traces for the Sliding Window Cache

The \( Q\)-SARSA(\( \lambda \)) algorithm use eligibility traces, but eligibility traces for function approximation is not as straight forward as for the tabular case, and the implementation is different depending on which function approximator is used. For an incrementally trained neural network, eligibility traces is a bit more complex than
for the tabular case, but the calculation overhead will usually be lower. Instead of maintaining an eligibility trace for each prior state-action pair, an eligibility trace is maintained for each weight in the network. The eligibility trace of a weight is an indication of how eligible the weight is for being updated, and it is calculated on the basis of the gradient for the weight. Sutton and Barto (1998) gives a detailed description of how these traces are incorporated into the SARSA(\(\lambda\)) algorithm, and also describes some of the challenges faced when implementing replacing eligibility traces into this model. This approach can unfortunately not be used with a batch training algorithm. The reason for this is that batch training algorithms do not calculate their gradients based on the current reward, but on the rewards in their batches, so the current reward can not be propagated back to the prior state-action pairs using a eligibility trace for the weights.

Batch algorithms have the advantage that they have a cache and as long as all the state-action pairs \((s, a)\) which have an eligibility trace \(e(s, a) \neq 0\) is located in the cache, batch algorithms can calculate the eligibility trace inside the cache. When the eligibility trace is calculated inside the cache, the eligibility trace is calculated in exactly the same way as tabular eligibility traces, which also means that calculating replacing eligibility traces can be done in the same way.

Vamplev and Ollington (2005) and Kamenetsky (2005) uses an on-line approach and they have implemented eligibility traces by maintaining an eligibility trace for each weight. The other combinations of Cascade-Correlation and reinforcement learning were not able use this approach because they used batch training, and they decided not to use any kind of eligibility trace.

5.5 Neural Fitted Q Iteration

Although there have not been any examples of batch training algorithms in combination with reinforcement learning before Rivest and Precup (2003), there has been at least one example after. Neural Fitted Q Iteration (NFQ) is a novel model-free reinforcement learning algorithm which combines reinforcement learning with RPROP training.

Riedmiller (2005) introduced Neural Fitted Q Iteration, which is a modification of the tree based Fitted Q Iteration presented by Ernst et al. (2005). The approach differs from the other batch approaches in a number of minor ways:

- It stores all experiences in a history window, instead of only the last \(n\) experiences.
- It allows for the neural network to be trained for several iterations without adding new experiences to its history window.
- The history window is not used as a lookup cache.

These differences are however only variations of the other approaches. What distinguishes NFQ from the other approaches, is the fact that it does not store the \(Q(s, a)\) values in the history window, but recalculates them as needed. I will annotate the calculated \(Q\) values as \(Q[s, a]\), so as not to confuse them with the \(Q(s, a)\) function, which is represented by the neural network.

In order for NFQ to calculate the \(Q[s, a]\) values, it stores the state \(s\), the action \(a\), the received reward \(r\) and the next state \(s'\) in its history window. When the NFQ algorithm trains its neural network, it uses the \(<s, a, r, s'>\) tuples in the history window to calculate \(Q[s, a]\) values and generate a training data-set. The \(Q[s, a]\) values, which are used for training are calculated by equation (5.5.1), which is essentially the standard Q-learning update rule except that it does not have a learning rate:
5.5. NEURAL FITTED Q ITERATION

\[ Q[s, a] \leftarrow r + \gamma \max_{a' \in A(s')} Q(s', a') \] (5.5.1)

The learning rate in the traditional Q-learning function, is used to make sure that previously learned experiences are not forgotten. This feature is not needed in NFQ, because it always trains using the complete history, so no prior experience is ever forgotten.

5.5.1 Enhancing Neural Fitted Q Iteration

Neural Fitted Q Iteration is closely related to Q-learning. Both use a similar update rule and both are model-free off-policy learning methods. The major difference is that NFQ recalculates the \( Q[s, a] \) values before training the neural network. With so many similarities between Q-learning and NFQ, it would be interesting to explore whether any of the modifications, which have been made to the Q-learning algorithm, can also be made to the NFQ algorithm.

NFQ is still a young algorithm, so it has not yet been combined with some of the more advanced features from Q-learning, and to the best of my knowledge, none of the enhancements that I suggest, have been suggested before.

The obvious modification to NFQ is to make it into an on-policy algorithm, so that you would have a Neural Fitted SARSA Iteration algorithm, or perhaps a Neural Fitted Q-SARSA Iteration algorithm. Both of these modifications are quite simple, since the update rule is so close to the original Q-learning update rule.

In order for an on-policy algorithm to be implemented for NFQ, the next action \( a' \) will need to be saved to the history window, so that the complete \( < s, a, r, s'>, a' > \) tuple is saved. With this tuple in the history, it is easy to define a Neural Fitted SARSA Iteration (NF-SARSA) update rule:

\[ Q[s, a] \leftarrow r + \gamma Q(s', a') \] (5.5.2)

Likewise the Neural Fitted Q-SARSA Iteration (NFQ-SARSA) update rule can be defined:

\[ Q[s, a] \leftarrow r + \gamma \left( (1 - \sigma) \max_{a'' \in A(s')} Q(s', a'') + \sigma Q(s', a') \right) \] (5.5.3)

Much like the Q-SARSA update rule, the NFQ-SARSA update rule can be converted into the standard NFQ rule by setting \( \sigma \) to zero and into NF-SARSA by setting \( \sigma \) to one.

If the NFQ-SARSA algorithm is to run for an extended amount of time, it will run very slowly, because the history window will be very large. A simple method for overcoming this problem, is to set a maximum size on the history window, and then discard the oldest tuples when new are inserted. This reintroduces the problem of forgetting, which was eliminated because the full history was available. This problem could simply be disregarded, because the problem will not be severe, since the large history window ensures that some amount of data is remembered. Another method for handling the problem, could be to reintroduce the learning rate \( \alpha \):

\[ Q[s, a] \leftarrow Q(s, a) + \alpha \left( r + \gamma \left( (1 - \sigma) \max_{a'' \in A(s')} Q(s', a'') + \sigma Q(s', a') \right) \right) - Q(s, a) \] (5.5.4)

Where a learning rate of one would be the same as not having introduced it at all, and likewise setting a maximum size of the history window to infinity, will be the same as not having introduced the maximum size.
It has been seen that the update rule for NFQ could be altered to resemble the Q-SARSA update rule, so the obvious next question is whether it is possible to add eligibility traces to NFQ-SARSA?

Eligibility traces for the batch version of Q-SARSA($\lambda$) is calculated inside the cache, for NFQ-SARSA only a history window exists and since $Q(s, a)$ values are not saved in this history window, it is not possible to calculate eligibility traces inside the history window. However, when the actual training data is generated, the $Q[s, a]$ values are calculated and it is possible to calculate eligibility traces on the basis of this, just like for the normal cache. In order to implement this for episodic tasks, it should however be possible to identify goal states within the history window, so that eligibility traces from one episode does not interfere with eligibility traces from other episodes.

With these modifications in place, the NFQ-SARSA($\lambda$) algorithm is defined, as an enhanced algorithm based on the NFQ algorithm. All of the modifications can still be disabled by setting the parameters for the NFQ-SARSA($\lambda$) algorithm correct. The full NFQ-SARSA($\lambda$) algorithm has been implemented and is benchmarked against the batch Q-SARSA($\lambda$) algorithm in chapter 6.

5.5.2 Comparing NFQ-SARSA($\lambda$) With Q-SARSA($\lambda$)

The major difference between NFQ-SARSA($\lambda$) and batch Q-SARSA($\lambda$) is the fact that NFQ-SARSA($\lambda$) recalculates the $Q[s, a]$ values before it trains the ANN, while Q-SARSA($\lambda$) uses the $Q(s, a)$ calculated when the state-action pair $(s, a)$ was visited.

There are three advantages of using batch Q-SARSA($\lambda$):

- Inserting a new $Q(s, a)$ value in the cache only changes the value for one state-action pair, which means that the training data for the neural network does not change so rapidly. The fact that the training data does not change so rapidly, should help the neural network to converge faster. There is, however, a chance that the function it converges toward is not the correct function, since it relies on information which was obtained at an earlier time-step ($t - k$), where the policy $\pi_{t - k}$ was not the same as the current policy $\pi_t$.

- The sliding window cache can be used directly as a lookup cache, which means that frequently accessed state-action pairs does not need to rely so much on the generalization capabilities of the neural network.

- Because it does not need to recalculate the $Q[s, a]$ values, it is much faster than the NFQ algorithm.

A major disadvantage of Q-SARSA($\lambda$) is, that the oldest $Q(s, a)$ values were calculated on the basis of a function approximator and a policy, which may have changed a lot since it was calculated. This means that the $Q(s, a)$ value in the cache may be a very poor approximation to the current $Q^\pi(s, a)$ value. This essentially means that $Q(s, a)$ values that are too old are of no use.

The advantages of the NFQ-SARSA($\lambda$) algorithm are:

- Since only the immediate reward is stored, the values that the NFQ-SARSA($\lambda$) algorithm stores are never outdated, and if the environment is a Markov environment, experience gathered in the first episodes are just as important as the experience gathered in the last episodes. This means that the algorithm can make very effective use of the experience it gathers.

- The algorithm can combine prior gathered information to give information about policies which have not been followed. Earlier, the grid-world problem
5.5. **NEURAL FITTED Q ITERATION**

from figure 4.4 on page 73, was used as an example of a problem where a model-based approach was able to find the optimal solution from only two runs, where a model-free approach would not. The NFQ-SARSA(\(\lambda\)) algorithm is the exception. This model-free approach would actually also be able to find the optimal solution from only two runs. When the NFQ-SARSA(\(\lambda\)) algorithm has executed the two runs its history window will include one tuple where the reward is 10, one where it is -10 and the rest of the included tuples will have a reward of zero. The tuples in the history window show the paths travelled in the two runs. As the NFQ-SARSA(\(\lambda\)) algorithm trains the ANN based on the two runs, the non-zero rewards will propagate back through the paths and will be expressed in the \(Q[s, a]\) values along the paths. The non-zero rewards will, however, also be propagated back through the optimal path, since the RPROP training algorithm will look globally at the tuples without considering the original order of the tuples. This will mean that the NFQ-SARSA(\(\lambda\)) will find the optimal path from only looking at the two runs.

The major disadvantage of the NFQ-SARSA(\(\lambda\)) is speed, because it requires that the \(Q[s, a]\) values are recalculated before training the ANN. This is especially a problem if there are many available actions in each state, since an execution of the ANN is required for each of these states. The NF-SARSA(\(\lambda\)) algorithm does not have this problem, but it is still far slower than the \(Q\)-SARSA(\(\lambda\)) algorithm.

Riedmiller (2005) shows good results for simple problems using the NFQ algorithm, and Kalyanakrishnan and Stone (2007) show that the algorithm can also be successful for the more complex domain of RoboCup keepaway soccer. Kalyanakrishnan and Stone (2007) compares the algorithm to a traditional on-line algorithm, and the experience replay algorithm (Lin, 1992). Their results shows that the NFQ algorithm uses the gathered experience much more efficient than the traditional on-line approach, and it shows that the experience replay algorithm produces results that are comparable to the NFQ algorithm. Kalyanakrishnan and Stone (2007) uses incremental training as batch training algorithm, for both NFQ and experience replay, so it is hard to directly make any conclusions on the basis of their experiments. However, they seem to suggest that the overall performance of NFQ and batch \(Q\)-learning are comparable, with no clear advantage to any of the algorithms. Kalyanakrishnan and Stone (2007) also show that NFQ and experience replay algorithms using neural networks consistently perform better than the same algorithms using CMAC.

The enhancements in the NFQ-SARSA(\(\lambda\)) algorithm, that are not available in the NFQ algorithm, should generally be seen as an advantage. However, there is a chance that the enhancements may degenerate some of the advantages of the original NFQ algorithm. The global look at the gathered experience, is what enables the NFQ algorithm to solve problems that other model-free methods are not able to solve. This global look is degenerated by the eligibility trace which only propagates reward down the path that is actually taken, and by the SARSA element of the NFQ-SARSA(\(\lambda\)) algorithm, which also favors the path that is taken, instead of the optimal path. The fact that it is possible to favor the paths that have actually been taken, as oppose to only looking globally may prove to be an advantage, but it may also prove to be a disadvantage. The experiments in chapter 6 shows whether the NFQ-SARSA(\(\lambda\)) algorithm is able to improve on the standard NFQ algorithm and appendix D will give a detailed discussion of the performance of the individual enhancements.
5.6 Reinforcement Learning and Cascading Networks

The sliding window cache, $Q$-SARSA($\lambda$) and NFQ-SARSA($\lambda$) could be used with a number of batch training algorithm for ANNs, like batch back-propagation, RPROP and Quickprop, but the really interesting property is the combination with constructive algorithms like Cascade-Correlation and Cascade 2. However, this combination raises a number of questions which are not raised with the standard batch and incremental algorithms.

The most important of them being, how should an algorithm add neurons to the ANN, when the function that it is trying to approximate is constantly changing? This is especially important since the input weights are frozen once the neuron is installed. If a neuron is installed at an early stage, and it produces information which is not valid for the current function, then the neuron will be of no benefit, and it will make it harder for later neurons to produce good results. This problem was also a concern in section 2.5.2 for Cascade-Correlation trying to learn a static function, but when the function is not static, the problem is even more pressing.

This problem is closely related to the problem of whether it is a good thing that the function approximator used in conjunction with reinforcement learning forgets earlier experiences as it learns new experiences. Rivest and Precup (2003) operates with two terms, small forgetting and catastrophic forgetting. Small forgetting is generally a good thing, because the $Q(s, a)$ function changes over time. Catastrophic forgetting is however not a good idea, since it will mean that the ANN forgets $Q(s, a)$ values for many areas of the state-action space, simply because they are not part of the cache at the moment. Due to the weight freezing, Cascade-Correlation does not suffer as much of forgetting, which is a good way of avoiding catastrophic forgetting, but it also discourages small forgetting.

Another concern is that reinforcement learning may take a long time, before the $Q(s, a)$ values are stable, which will also mean that the ANN will be trained for a long time, this long training may lead to many candidate neurons being added to the ANN, which is generally not desirable. The experiments made by Kamenetsky (2005) and Bellemare et al. (2004) in scaling reinforcement learning with Cascade-Correlation to larger problems are a bit discouraging, and the experiments do not perform well for large problems. They seem to add many neurons in the beginning of the training, whereafter the learning slows down considerably. This seems to indicate that they suffer from the weight freezing problem, and an inability to forget the learning from the initial training. Furthermore, it could indicate that the neurons are added too quickly, since most of the neurons are added before the $Q(s, a)$ values become stable.

These concerns about using Cascade-Correlation with reinforcement learning, leads to the need for a strategy that ensures that the ANN does not grow too fast, and that the ANN can exhibit some degree of forgetting. The standard Cascade 2 algorithm uses a patience parameter which is meant to ensure that over-fitting does not occur. Bellemare (2006) combined this approach with a minimum number of epochs, to ensure that a reasonable amount of training will actually happen before a switch is made from training the outputs to training new candidates. This helps ensuring that the network does not grow too fast, and because the ANN is smaller it will also be easier for the ANN to exhibit small forgetting.

In order to implement even more forgetting and eliminate some of the weight freezing problem, I propose the solution which was also proposed to solve the weight freezing problem in section 2.5.3. The proposed solution is to train the whole ANN after the outputs have been trained alone. This will enable some amount of forgetting, but it might reintroduce the moving target problem, so a lower learning rate
might be required when training the whole ANN.

The $Q$-SARSA($\lambda$) algorithm from algorithm 5 will be mostly unchanged when used in combination with Cascade-Correlation and the sliding window cache. The primary change will be that the $Q(s,a)$ values will be read from the cache and that there will be introduced a train cache step after each time-step. The full $Q$-SARSA($\lambda$) with whole ANN training is illustrated in algorithm 7.

Algorithm 7 The $Q$-SARSA($\lambda$) algorithm combined with Cascade-Correlation and whole ANN training.

\begin{verbatim}
Q ← empty sliding window cache
training-phase ← output-training
for all episodes do
  for all $s \in S$ and all $a \in A(s)$ do
    $e(s,a) ← 0$
  end for
  $s ←$ some start state
  $a ← \pi(s)$, where $\pi$ is a policy based on $Q$ (e.g. Max-Boltzmann)
while $s$ is not a terminal state do
  Take action $a$ and observe reward $r$ and next state $s'$
  $a' ← \pi(s')$
  $\delta ← \alpha \left( r + \gamma \left( (1-\sigma) \max_{a'' \in A(s')} Q(s',a'') + \sigma Q(s',a') \right) - Q(s,a) \right)$
  $e(s,a) ← 1$
  for all $a'' \in A(s)$ do
    $e(s,a'') ← 0$
  end for
  for all $s'' \in S$ and all $a'' \in A(s'')$ do
    $Q(s'',a'') ← Q(s'',a'') + e(s'',a'')\delta$
    $e(s'',a'') ← \gamma \lambda e(s'',a'')$
  end for
  $s ← s'$
  $a ← a'$
if cache is filled then
  if training-phase = output-training then
    Train outputs for one epoch
  if output training has stagnated then
    training-phase ← whole-ann-training
  end if
else if training-phase = whole-ann-training then
  Train the whole network for one epoch
if training has stagnated then
  Train candidates and install a candidate in the network
  training-phase ← output-training
end if
end if
end if
end while
end for
\end{verbatim}
5.7 Reinforcement Learning Implementation

The Cascade 2 algorithm is implemented as part of the open source C library FANN, so that the implementation can be used by other developers. Implementing functionality in an existing C library requires that a lot of thought is put into the implementation, and the interfaces exposed by the implementation. It also requires that the implementation is thoroughly documented, and that the coding guidelines for the library is maintained. For the FANN library this means, that all code must be written in C, and that no external libraries are used. For the Cascade 2 implementation this posed quite a deal of challenges, especially due to the dynamic memory allocation which is needed by the algorithm.

Since the FANN library is an ANN library, and not a reinforcement learning library, the reinforcement learning implementation need not be integrated into the library, and the implementation need not comply to the strict requirements of the FANN library. For this reason the reinforcement learning implementation has been implemented as a separate C++ program, which use the FANN library as the ANN implementation.

The reinforcement learning implementation supports the full $Q$-SARSA($\lambda$) algorithm with several kinds of function approximators; a simple tabular lookup table, a standard on-line incrementally trained ANN, an RPROP trained ANN and a Cascade 2 trained ANN, where the latter two uses the sliding window cache. However, since eligibility traces are implemented directly in the cache, the incrementally trained ANN is not able to utilize this functionality. The implementation also supports the NFQ-SARSA($\lambda$) algorithm in combination with RPROP or Cascade 2. When the Cascade 2 algorithm is used, it is possible to allow the whole ANN to be trained, as described in algorithm 7. It is also possible to specify a minimum and maximum number of steps, that there should be between adding neurons to the ANN, and it is possible to specify a desired MSE value, where no neurons is added as long as the MSE is below this value.

The implementation supports two different rehearsing strategies, either no rehearsing, or full rehearsing where all possible actions are rehearsed. Section 4.3 describes a number of different exploration strategies, but as stated in section 4.6.1 only a few of these can be used in combination with function approximators. For this reason only $\epsilon$-greedy and Max-Boltzmann exploration is implemented. These selection strategies can be combined with a tabu-list, which ensures that the same explorative state-action pair is not selected twice. To allow for more exploration in the beginning of the learning, and less towards the end, it is possible to anneal $\epsilon$ and the Boltzmann temperature. Likewise it is also possible to anneal the learning rate $\alpha$.

The reinforcement learning implementation is made to interface with the RL-Glue benchmarking framework (RL-Glue, 2005; White, 2006), so that benchmarking the implementation against various standard problems will be easy. To further ease the benchmarking, various key data is recorded during learning, and the entire agent is saved to a file from time to time, so that both the agents on-line and off-line performance can be evaluated. All of the parameters for the algorithms can be controlled from the command-line, to support easy benchmarking of the parameters, and to make it easier to reproduce the benchmarks. Appendix E includes a detailed description of how the implementation should be compiled and a detailed description of the command line parameters.
Chapter 6

Reinforcement Learning Tests

This chapter will test the $Q$-SARSA($\lambda$) algorithm alone and in combination with both incrementally trained, batch trained and cascade trained neural networks. In addition to this the $Q$-SARSA($\lambda$) algorithm will also be compared to the NFQ-SARSA($\lambda$) algorithm. Since scalability is the primary reason for including neural networks, scalability will be an important part of the test and the algorithms will be tested on small as well as large problems. Section 6.1 will focus on selecting test problems to be used for the tests, while section 6.2 will focus on how the tests should be carried out.

Throughout this chapter there will be many graphs showing performance of the algorithms. To conserve space many of these graphs have been kept small in this chapter, but enlarged versions are available in appendix G.

6.1 Reinforcement Learning Test Problems

For the function approximation problems in section 3.2 on page 31, it was important to specify which classes of problems that existed, so that the benchmarks could be made as exhaustive as possible. For reinforcement learning this is even more important, since the main focus of this thesis is the performance of the implemented reinforcement learning algorithms. Traditionally, not much work has been done trying to classify which kinds of reinforcement learning problems exist, and they have traditionally only been classified as episodic/non-episodic and discrete/continuous. Riedmiller (2005) takes this classification a bit further and identifies three classes of problems:

**Goal reaching** is a standard episodic task like e.g. the mountain car (see section 6.1.2) problem, where the agent is required to reach some kind of goal, and where the episode ends when the goal is reached.

**Regulator problems** is a non-episodic controller problem, where the agent is required to be kept in a certain region of the state space.

**Avoidance control** is a problem like e.g. cart pole (see section 6.1.3), which requires that the agent is kept in a “valid” region, and where the episode is ended when the agent exits this region.

Many of the traditional reinforcement learning problems belong to one of these three classes, and these three classes can serve as basis for a more exhaustive list of classes, along with the discrete/continuous classification.
The distinction between discrete and continuous is also a distinction on the size of the state-action space. The size of the state-action space is an important factor when determining how difficult a problem is to solve, but the difficulty cannot simply be measured in the number of state-action pairs, especially since this number is infinite for the continuous case. An important factor is how uniformly the $Q^*(s,a)$ values are distributed throughout the state-action space. If the values are very uniformly distributed, a problem with many or infinite state-action pairs, may be easier to solve than a problem which has fewer state-action pairs, but a very non-uniform distribution in the $Q^*(s,a)$ values.

Avoidance control and regulator problems are essentially the episodic and non-episodic versions of the same problem, which can be referred to as a controller problem. In contrast to the controller problems, there are the goal reaching problems, where the agent has to reach some kind of positive reward goal state. Some goal reaching problems also have negative reward goal states which must be avoided, and a special case of these problems are the two player game problems, where the agent has to reach a goal, while avoiding that the opponent reaches his goal. One of the special properties of problems where there are negative reward goals, is that exploration is usually more expensive than for standard goal problems. This is also the case for two player game problems, where too much exploration may lead to the opponent winning many of the games, and this may hinder the learning. Many two player game problems has the special property, that you can predict the after-state which occurs immediately after you have taken an action. This means that instead of using the $Q(s,a)$ function, the $V(s)$ function can be used for each of these after-states, and the action for the after-state with the highest $V(s)$ value can be chosen. Since the input to the $V(s)$ function is only a state, and not a state and an action, this function is simpler to approximate, and the learning algorithm can also be simplified a bit.

The episodic goal reaching problems can be separated into two groups; the problems where a positive or negative reward goal will always be reached, no matter which policy is chosen, and the problems where a policy can lead to an infinity loop. If a policy can lead to an infinity loop, it is important that learning occurs during the episodes, and not just after the episodes. In practical cases, there are usually some kind of exploration factor, so the policy is not fixed, and the agent will always reach a goal even if there is no training during the episode. It may, however, still be a problem that the policy can be degenerated to a degree, where it is very difficult to reach a goal. In this case the learning may be hindered dramatically. The reason for this, is that a very long episode may occur when the agent has problems reaching the goal. This long episode will generate many learning samples which have very little learning value since they do not find the goal, and when the agent tries to learn from these samples, it will have a tendency to forget earlier learned experiences. For problems where the goal is always reached within a limited number of steps, such problems does not occur. These kinds of problems are also easier to solve for algorithms that use batch trained neural networks, because you can guarantee that there will always be a certain number of episodes in the cache. The same problem also exists for the avoidance control problems, since successful episodes may be very long. However, the problem is not as significant, since the policy followed is close to the optimal for the avoidance control problems, while it is far from optimal for the goal reaching problems.

As can be seen by this discussion, episodic avoidance control and especially goal reaching problems can be split into many sub-categories, and there are many features of the problems that can make them more or less simple to learn. There are also a number of sub-categories which exists for both episodic and non-episodic problems. The discrete/continuous distinction being one of the more important, because it distinguishes which kind of algorithm and function approximator that can
be used. If the actions themselves are also continuous, it will be impossible to solve
the problem for standard reinforcement learning algorithms. For this reason this
category is usually left out, but there exists another distinction on the actions; the
problems where the available actions are fixed, and the problems where the available
actions are dependent on the state. Two player games are often an example of the
latter, and although the algorithms themselves are usually not concerned too much
about the number of actions, it may be easier for an algorithm to learn problems
with a small fixed number of actions. In these situations the agent can learn that
some actions are usually better than other, which may be used as a guideline in
unknown areas of the state space.

Another feature of the problem is how stochastic the problem is, and how evenly
the stochasticity is distributed throughout the state-action space. Generally speak-
ing it is easier to learn a low evenly distributed stochasticity, than a high non-
evenly distributed stochasticity. Some amount of stochasticity may, however, help
the training algorithm reach more parts of the state-action space, without so much
exploration. The stochasticity added to backgammon by means of the dice, is ac-
tually stated as one of the primary reasons that TD-Gammon by Tesauro (1995)
had such a huge success. Had the game not included a dice, it would be difficult to
reach large parts of the state space.

Ideally when choosing test scenarios for a reinforcement learning algorithm, all
of the different kinds of problems should be included as test scenarios. It is, however,
not a simple task to set up a test-scenario, and the tests may also be complicated
and time consuming to perform. For this reason only a limited number of problems
are used when testing reinforcement learning algorithms. When selecting
problems for testing the variations of the $Q$-SARSA($\lambda$) algorithm, I have used the
following criteria:

- The problems should span many of the mentioned categories and sub-categories
  of problems. With particular emphasis on including:
  - Problems with small and large state-action spaces
  - Discrete and continuous problems
  - Goal reaching problems and avoidance control problems

- The problems should be well documented in literature, to make comparison
  with other algorithms easier.

I have purposely not included non-episodic problems, because they are not so
widely referred in literature and because most research in reinforcement learning
have focussed on episodic problems. I ended up selecting four different problems,
which I feel fulfil the selection criteria: blackjack, mountain car, cart pole and
backgammon. Blackjack is a simple discrete goal reaching problem, mountain car
is a semi complicated continuous goal reaching problem and cart pole is a semi
complicated continuous avoidance control problem. All of these three problems are
implemented in the RL-Glue framework (RL-Glue, 2005; White, 2006), and have
either been taken directly from RL-Glue, or from the “Reinforcement Learning
Benchmarks and Bake-offs II” workshop at the NIPS 2005 conference, where RL-
Glue was used. The backgammon problem is a large complicated discrete two player
game problem, with a variable number of available actions ranging from a single
available action up to several hundred actions. The implementation is the same
implementation as used by Bellemare et al. (2004), that I have altered for it to be
incorporated into the RL-Glue framework.

These four problems are summarized in table 6.1 and will be described in sec-
tion 6.1.1 to 6.1.4. Section 6.2 to 6.7 will test the different algorithms on the first
three problems, while the backgammon problem will be handled separately in section 6.8 because it is such a large and complex problem which can be solved in several different ways.

<table>
<thead>
<tr>
<th>Problem type</th>
<th>Blackjack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackjack Problem type</td>
<td>Episodic goal reaching problem, where the goal is always reached after a few steps.</td>
</tr>
<tr>
<td>Problem size</td>
<td>Discrete problem with 720 state action pairs.</td>
</tr>
<tr>
<td>Stochasticity</td>
<td>Stochasticity provided in each step by the random cards.</td>
</tr>
<tr>
<td>Static / Dynamic</td>
<td>The environment is static.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem type</th>
<th>Mountain Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mountain Car Problem type</td>
<td>Episodic goal reaching problem, with a hard to find goal, and no guarantee of reaching the goal.</td>
</tr>
<tr>
<td>Problem size</td>
<td>Continuous problem with a relatively smooth $V(s)$ function, for most parts of the state space.</td>
</tr>
<tr>
<td>Stochasticity</td>
<td>Stochasticity provided by a random starting position.</td>
</tr>
<tr>
<td>Static / Dynamic</td>
<td>The environment is static.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem type</th>
<th>Cart Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cart Pole Problem type</td>
<td>Episodic controller problem where an optimal policy will lead to a newer ending episode.</td>
</tr>
<tr>
<td>Problem size</td>
<td>Continuous problem with a relatively small valid region and many possible actions.</td>
</tr>
<tr>
<td>Stochasticity</td>
<td>Stochasticity provided by a random starting position.</td>
</tr>
<tr>
<td>Static / Dynamic</td>
<td>The environment is static.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem type</th>
<th>Backgammon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backgammon Problem type</td>
<td>Episodic goal reaching problem, where the goal is always reached (typically after 20-60 episodes, but much longer episodes may also occur).</td>
</tr>
<tr>
<td>Problem size</td>
<td>Discrete problem with an estimated $10^{20}$ different states and between one and several hundred possible actions in each state.</td>
</tr>
<tr>
<td>Stochasticity</td>
<td>Stochasticity provided in each step by the roll of the dices.</td>
</tr>
<tr>
<td>Static / Dynamic</td>
<td>If backgammon is played against a static opponent, the problem is also static, but if self-play is used, the environment is very dynamic and essentially changes after each step.</td>
</tr>
</tbody>
</table>

Table 6.1: A summary of the four problems which is used for testing the $Q$-SARSA($\lambda$) algorithm. The easiest problem blackjack is listed first, then the two intermediate problems mountain car and cart pole and lastly the difficult problem of backgammon.

### 6.1.1 The Blackjack Problem

The blackjack problem is the problem of playing one hand of blackjack against a dealer, using rules that are a bit more simple than the rules used at casinos. The
player is dealt two cards, and the dealer is dealt one card face down and one card face up. The objective for the player, is to obtain cards that sum to a value which is higher than that of the dealer, without exceeding 21. To obtain this goal, the player has two available actions, hit or stand, where hit gives one more card and stand ends the flow of cards. If at this point the player has exceeded 21, the game is ended and the player has lost, otherwise the dealer turns over his face down card, and gets more cards, until his cards sum to 17 or more. If the value of the dealers cards exceeds 21 he will have lost, otherwise the winner is determined on the basis of the value of the cards. If the player has the highest value he wins, and if the dealer has the highest value he wins. If both the dealer and the player has the same value the game is a draw, except in the case where both player and dealer has 21. In this case the player will win, if he received 21 on the first two cards (also referred to as a natural), and the dealer did not.

The cards dealt at casinos are dealt from a limited deck of cards, and this means that a skilled player can gain an advantage by knowing exactly which cards that have already been dealt. This means that the casino version of blackjack is not a Markov decision process. In order to make the blackjack problem into a MDP, the cards in this problem is dealt from an infinite deck of cards.

The problem is taken from Sutton and Barto (1998) with the slight modification that it is possible to hit when you have a natural (21 on the initial two cards), and that the player is given a choice even though the value of his cards are below 12 and there is no possibility of exceeding 21 by hitting. The state space consists of 3 discrete variables; the current value of the players hand \([4\rightarrow 21]\), the value of the dealers face-up card \([2\rightarrow 11]\), and a binary indication of whether the player has a usable ace. The action space consists of a binary variable indicating whether the player hits or stands. The starting state, is the state where the player has two random cards and the dealer has one random face-up card, and the episode ends when the player stands, or the value of the players cards exceeds 21. The reward for all steps that do not end the episode is 0, the reward for a win is 1, -1 for a loss and 0 for a draw.

The blackjack problem is a goal reaching problem, which is mentioned in many different variations throughout the literature, so comparison with literature might be a bit difficult. This particular implementation is, however, taken from the RL-Glue (2005) library, and it is possible to compare results directly with the results published for RL-Glue.

The total number of states is 360, and the total number of state-action pairs is 720. The low number of state-action pairs makes this problem ideal for a tabular solution.

6.1.2 The Mountain Car Problem

The mountain car problem is the problem of driving an under-powered car up a mountain road, as illustrated in figure 6.1. The difficulty of the problem lies in the fact that the car does not have enough power to accelerate up the road, so it will have to move away from the goal by backing up the left slope, before it can apply full throttle and reach the goal to the far right. Just like the blackjack problem, the mountain car problem is taken from Sutton and Barto (1998).

The mountain car problem has two continuous state variables; the cars position in the range from -1.2 to 0.5, and the cars velocity in the range from -0.7 to 0.7. There are three possible actions; full reverse throttle, full forward throttle and no throttle. The car starts at a random position within the -1.1 to 0.49 range with a velocity of 0.0. All steps that does not lead to the goal yields a reward of -1, and the physics of the car is modelled as described by Sutton and Barto (1998).
CHAPTER 6. REINFORCEMENT LEARNING TESTS

6.1.3 The Cart Pole Problem

The cart pole problem was introduced in chapter 1, and is the problem of balancing a pole on top of a cart that moves on a track, as illustrated in figure 6.2.

The objective is to balance the pole as long as possible, and preferably keep the pole close to vertical in the middle of the track. This objective is obtained by applying forces to the cart.

The cart pole problem has four continuous state variables: pole angle in radians from vertical, pole angular velocity in radians per second, cart position in meters from the center and cart velocity in meters per second. There are 21 discrete actions, corresponding to the discrete negative, zero and positive forces from -10 to 10 newton. The pole starts with a velocity of 0 and random pole position within the range $-\pi/18$ to $\pi/18$. The cart starts with a velocity of 0 and a position in the range -0.5 to 0.5. The episode ends if the cart moves off the track ($|\text{cart position}| \geq 2.4$) or the pole falls ($|\text{pole angle}| \geq \pi/6$).

The cart pole problem is just like the two other problems, a problem which can be found throughout the reinforcement learning literature, but it exists in many different variations, so comparison can be problematic. This version of the cart pole problem is an episodic avoidance control problem, but other variations are
specifed as non-episodic problems, where the pole is allowed to move around 360
degrees, and where the cart is blocked from moving off the track.

Like the mountain car problem, this implementation is also the same as used for
the NIPS 2005 workshop, and the episode is also ended after 300 steps. The reward
structure used for this workshop is -1000 if the pole falls ($|\text{pole angle}| \geq \pi/6$) or the
cart moves off the track ($|\text{cart position}| \geq 2.4$), 0 if the pole is completely balancing
on the middle of the track balancing ($|\text{pole angle}| \leq \pi/60$ and $|\text{cart position}| \leq
0.05$), and -1 otherwise. The physics of the cart and pole is modelled as described
in Selfridge et al. (1985), and parameters for the physics are; a cart mass of 1.0, a
pole mass of 0.1, a pole length of 0.5 and no friction.

The reward structure for the cart pole problem only consists of negative rewards
and since the episode is ended after 300 steps, an optimal policy for this problem
will move the pole into the region where it is completely balanced, and stay there
until the episode ends. If the optimal policy is followed, the reward for each episode
will be a small negative number, depending on how long it will take to get from the
initial position to the balanced position. However, if the pole is not balanced for the
full 300 steps and the pole falls, a negative reward of -1000 plus -1 for each of the
unbalanced steps will be given. This poses a problem, since a pole which falls after
299 steps will probably have spent more steps in an unbalanced state, than a pole
that falls immediately, which means the the pole that balances for 299 episodes will
also receive a higher negative reward. This problem made it very difficult for some
of the algorithms used at the workshop to learn how to balance the pole, and some
even changed the reward structure to be able to solve the problem. For this reason
I have also changed the reward structure, so that a reward of two is received when
the pole is completely balanced, a reward of -1000 is received when pole falls and a
reward of one is received for all other steps. This reward structure encourages the
agent to balance the pole for as long as possible.

6.1.4 The Backgammon Problem

The backgammon problem is the two player game problem of learning to play
backgammon. The problem was first treated in a reinforcement learning context by
Tesauro (1995) and has been tested on a combination of Cascade-Correlation and
reinforcement learning by Bellemare et al. (2004). Marc G. Bellemare has been so
kind as to let me use his implementation of the game for this thesis. The problem
follows the standard rules for single game backgammon, which also means that the
double cube, which is usually used in tournament play, is not used.

The backgammon problem is as such a discrete problem, since there is only a
limited number of states (board positions) but if the input encoding of Bellemare et al.
(2004) is used, continuous variables will also be present among the input variables.
These input variables can, however, only have a limited number of different values,
so the problem can still be considered a discrete problem. Using this encoding, the
state representation consists of 196 discrete variables. The actions are also discrete,
but the number of available actions in each step are not fixed, and there can be
between 1 and several hundred actions available in each step. The reward scheme
gives a reward of zero for all actions that does not end the game, and a reward of
1 or -1, respectively, for winning and loosing the game.

Since the problem is a two player game problem, after-states can be used, which
in effect means that the actions are not represented by how the checkers are moved,
but by the resulting board position after the checkers have been moved. For this
reason the $V(s)$ function is used instead of the $Q(s,a)$ function.

Learning to play backgammon, can either occur through playing against a fixed
player, or through self-play, where the agent plays both sides of the table. When
learning occurs through self-play, the traditional methods for measuring perfor-
mance, such as average reward does not make sense, since the agent will always win approximately 50% of the time when it plays against itself. A more important measure, is how well the final agent performs against some fixed player.

Even though the backgammon problem is not a continuous problem, and as such the state-action space is smaller than the mountain car and cart pole problems, the problem is more complex than any of the other problems. The problem is estimated to have \(10^{20}\) (Tesauro, 1995) possible states, and the \(V^*(s)\) values are not very uniformly distributed. Two similar board positions may have dramatically different \(V^*(s)\) values, if e.g. the one position has a checker which is likely to be hit\(^1\) in the next round, and the other does not.

The backgammon problem will primarily be used for testing the scalability of the \(Q\)-SARSA(\(\lambda\)) algorithm because it is such a large and complex problem. The backgammon problem will be covered separately in section 6.8, while the other three problems will be covered in section 6.2 to 6.7.

### 6.2 Reinforcement Learning Configurations

The following sections will cover testing of the implemented training algorithm on the blackjack, mountain car and cart pole problems. The test will be focussed on three distinct areas, where it will evaluate how:

- The \(Q\)-SARSA(\(\lambda\)) algorithm compares to the basic \(Q(\lambda)\) and SARSA(\(\lambda\)) algorithms.
- The combination of \(Q\)-SARSA(\(\lambda\)) with batch training algorithms and cascading algorithms compares to the incrementally trained \(Q\)-SARSA algorithm.
- The Neural Fitted \(Q\)-SARSA(\(\lambda\)) Iteration (NFQ-SARSA(\(\lambda\))) algorithm compares to the \(Q\)-SARSA(\(\lambda\)) algorithm.

The \(Q\)-SARSA(\(\lambda\)) algorithm will primarily be compared to the SARSA(\(\lambda\)) and \(Q(\lambda)\) algorithms on the blackjack problem, since this is a small discrete problem, where comparison can be made without including other factors such as the neural network. The comparison will also be made for the mountain car, cart pole and backgammon problems, but the comparison will not be as thorough. In order to test the performance of the sliding window cache, the batch and cascading algorithms will be tested against the incremental algorithm for the mountain car and cart pole problems. The same problems will be used to test the NFQ-SARSA(\(\lambda\)) algorithm against the regular \(Q\)-SARSA(\(\lambda\)) algorithm. The backgammon problem which will be tested in section 6.8, will primarily be used to test how well the cascading algorithms scale, but it will also test the NFQ-SARSA(\(\lambda\)) algorithm and how the \(Q\)-SARSA(\(\lambda\)) algorithm compares to the \(Q(\lambda)\) and SARSA(\(\lambda\)).

The performance of the individual algorithms will primarily be compared to each other, but since the mountain car and cart pole problems are taken from the NIPS 2005 conference, their results will also be compared to the results from the conference. However, the algorithms at the NIPS conference are primarily highly optimized model-based and model-free algorithms, which are specifically targeted at the medium sized problems from the conference and many of the algorithms use knowledge of the problems to discretize them and hence make them simpler to solve. For this reason it can not be expected that the model-free algorithms in this thesis can produce comparable results. However, the algorithms in this thesis has

\(^1\)A hit in backgammon occurs when a player lands on a space, that is only occupied by one of the opponents checkers. In this case the opponents checker will be put on the bar.
the advantage that they are designed to scale to larger problem sizes, which is not
the case for the algorithms at the NIPS conference.

There are several different ways of measuring the performance of reinforcement
learning, and they generally split into two categories; measuring performance during
training and measuring performance after training, also referred to as the on-line
and off-line performance.

On-line performance is usually visualized as a graph of the cumulative reward
and the advantage of this approach is that it clearly shows how the performance
evolves over time. The cumulative reward is also often measured as a single number
after a fixed number of episodes. For easier comparison this number is often divided
by the number of episodes, to provide the average reward for the entire learning
run. The cumulative reward is usually displayed per episode, which means that
the number of steps needed to get the reward is not of any importance. For many
problems like blackjack and backgammon this is exactly what we want, but for e.g.
the mountain car problem all episodes that find the goal will have the same episode
reward, unless some negative reward is added to all steps that does not lead to
the goal. For these kinds of problems an often used measurement is the average
number of steps used to get to the goal. This measurement has the advantage
that it is independent of the reward structure, which allows problem formulations
with different reward structures to be easily compared. However, this is also the
disadvantage of this measurement, since it will not show the actual received reward,
which in turn can mean that it shows a false image of the performance.

Off-line performance is usually measured after a fixed number of episodes, as
the average reward received by running the agent off-line, where no exploration
and learning occurs, for a fixed number of episodes. This performance can also be
visualized as a graph by measuring the off-line performance several times during
the learning phase. The main advantage of this measurement is that it shows the
actual performance of the learned policy, without including any cost of exploration.
As mentioned in section 4.3 on page 52, off-line performance is not always the most
important measurement, since it may be desirable to obtain some level of reward
during training, and since slowly changing environments will require that learning
is maintained throughout the lifetime of the agent. For the problems tested in
this thesis, off-line performance is however an important measurement and for the
self-playing backgammon agent it will actually be the only available measurement.

The measurement, which will be used when evaluating the different algorithms,
is a combination of off-line and on-line performance, with most emphasis on off-
line performance. For general trends, the on-line and off-line performance will only
be reported after a fixed number of episodes, but for more in-depth analysis, the
performance will be displayed as a graph over the episodes. When the algorithms
are tested on the individual problems, the tests will focus on the most interesting
aspects of the combination of algorithm and problem, instead of tediously going
through all the parameters. This means that the same aspects and parameters will
not be the focus of all the tests, and that the aspects and parameters which are in
focus will receive more attention.

There are a number of different parameters for the Q-SARSA(λ) and NFQ-
SARSA(λ) algorithms, that will receive focus during the tests. These parameters
are described in detail in chapter 4, but table 6.2 provide a quick overview.

With all the possible parameters for the tests, the parameters could be tuned in
many different ways and the results could lead to different conclusions depending on
the tuning of the algorithm. For this reason, I have made sure that the parameters
are tuned in the same way for each of the different problems and algorithms. The
tuning method starts out with a random set of parameters, and for each of the
parameters a number of different values are tested, with the remaining parameters
fixed. This will create a graph for the on-line and off-line performance for each of
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>The learning rate parameter for the $Q$-SARSA($\lambda$) algorithm as a value from 0 to 1, where a value of 0 means that no learning will occur and a value of 1 means that earlier observation will be forgotten when new observations are made.</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Determines how often an explorative step is taken for $\epsilon$-greedy exploration as a value from 0 to 1, where a value of 0 means no exploration and a value of 1 means exploration in all steps.</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>Determines how much the future reward is discounted as a value from 0 to 1, where a value of 0 means that only the immediate reward will be considered, and a value of 1 means that all future rewards will be considered equally.</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>The parameter controlling the eligibility traces as a value from 0 to 1. The eligibility traces are also dependent on the $\gamma$ parameter, so a $(\gamma \ast \lambda)$ value of 0 means no eligibility trace and a $(\gamma \ast \lambda)$ value of 1 means that all prior steps will receive the full reward.</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Determines how $Q(\lambda)$ and SARSA($\lambda$) are combined to form the $Q$-SARSA($\lambda$) algorithm as a value from 0 to 1, where a value of 0 means that the $Q(\lambda)$ algorithm is used and a value of 1 means that the SARSA($\lambda$) algorithm is used.</td>
</tr>
<tr>
<td>Cache size</td>
<td>The size of the cache used for the sliding window cache and the NFQ-SARSA($\lambda$) algorithm.</td>
</tr>
<tr>
<td>Cache commit interval</td>
<td>The number of cache writes between the neural network is trained with the cache.</td>
</tr>
</tbody>
</table>

Table 6.2: A summary of the primary parameters that will be tuned during the reinforcement learning tests.

The parameters. By looking at these graphs it is possible to see which parameters, that could benefit from being altered. One or more of the parameters are altered after which a new test will be run and a new set of graphs will be created. This tuning will continue until the graphs converge and show that no benefit can be made from altering a single parameter. In order to make sure that the tests converges and that they are reproducible, the random number generator is seeded with a fixed value before each run. The process of tuning the parameters is time consuming, but relatively simple and it assures that the parameters for all the algorithms are tuned equally. However, there is no guarantee that the tuning will result in optimal parameters.

### 6.3 Tabular $Q$-SARSA($\lambda$)

Comparing $Q$-SARSA($\lambda$) to $Q(\lambda)$ and SARSA($\lambda$) requires that the effect of the $\sigma$ parameter for the $Q$-SARSA($\lambda$) algorithm is evaluated. This parameter will need to be evaluated under several different circumstances. The simplest form is the tabular form, where there is no underlying neural network. This form is also the purest form, since the algorithm is not dependent on any external factors. This simple form will give a basis for the evaluation of the $Q$-SARSA($\lambda$) algorithm against the $Q(\lambda)$ and SARSA($\lambda$) algorithms, but the results will need to be verified in more complex environments.

In order for a tabular algorithm to be used, the problem needs to be a discrete
6.3. TABULAR Q-SARSA(\(\lambda\))

problem. The only two discrete problems selected in section 6.1 are the blackjack problem and the backgammon problem. However, the backgammon problem is so complex that a tabular solution is not viable.

The blackjack environment implementation is taken from the RL-Glue library\(^2\), where Adam White reports that he can get an on-line performance with an average reward per episode of -0.1920, during the first 100,000 episodes, with a simple tabular SARSA(0) implementation. He also reports an off-line performance with an average reward per episode of -0.0441, after learning for 10,000,000 episodes. The average off-line reward was measured by running the agent for 100,000 episodes.

6.3.1 Tabular Q-SARSA(\(\lambda\)) for Blackjack

Before evaluating the \(\sigma\) parameter, appropriate values for the remaining parameters needs to be evaluated. For simplicity only \(\epsilon\)-greedy exploration is used, and no annealing is used. This leaves the \(\epsilon, \alpha, \lambda\) and \(\gamma\) parameters. A complete overview of the optimal parameters will require that all possible combinations of the parameters needs to be explored, and all of these combinations will also need to be combined with all possible values for \(\sigma\). Even if the parameters are only allowed 10 different values, the computation requirements are not feasible. This is, however, not as big a problem as it might seem. In order for a realistic evaluation of the \(\sigma\) parameter, the parameters need not be optimal, they just need to be suitable for evaluating the \(\sigma\) parameter. Suitable parameters are parameters that perform reasonable well, and that allow for some level of exploration, because if \(\epsilon\) is zero, then the value of \(\sigma\) will not matter.

The SARSA(0) implementation by Adam White uses the parameters \(\alpha = 0.001, \epsilon = 0.2\) and \(\gamma = 1.0\). By setting \(\sigma\) to 1.0 and \(\lambda\) to 0.0, the Q-SARSA(\(\lambda\)) algorithm can be made to resemble the SARSA(0) implementation by Adam White. Adam’s implementation also used an optimism in the face of uncertainty method, by setting \(Q(s, a)\) to 1.0 for all state-action pairs that have not yet been visited. This behavior was also reproduced in the Q-SARSA(\(\lambda\)) implementation, and a test confirmed that the Q-SARSA(\(\lambda\)) implementation was able to produce results that were similar to the results by Adam. The average reward for the on-line performance was -0.2030, during the first 100,000 episodes, and the average reward for the off-line performance was -0.04457 after 10,000,000 episodes. When the parameters for the Q-SARSA(\(\lambda\)) algorithm was optimized for a \(\sigma\) value of 0.5 as described later on in this section, the on-line performance was significantly improved to -0.1344, and the off-line performance was improved to -0.03963, which is a significant improvement compared to the results of Adam White. The primary focus of this section is, however, not the comparison with other results, but the evaluation of the \(\sigma\) parameter. The remainder of this section will focus on how these parameters are optimized, and the results will be used to see whether the \(\sigma\) value can be used to improve the learning.

In order to find suitable parameters, the on-line and off-line performance for each of the parameters are evaluated, while keeping the remaining parameters fixed. The parameters are tuned iteratively as described in section 6.2. During the test of each of the parameters, both the on-line and off-line performance is evaluated. To make sure that the results are a true evaluation of the performance, and not influenced by the order that the random cards are received, the random number generator which produces the cards is seeded with a fixed value before the learning begins, and before the off-line performance is measured. By choosing this approach, the problem is changed a bit, since the cards are actually predictable, but since the reinforcement learning algorithm treats the problem as a MDP, I do not view this as

\(^2\)The RL-Glue library can be downloaded from http://rlai.cs.ualberta.ca/RLBB/top.html
a problem, especially since 100,000 episodes is used for evaluating the performance. The alternative to this, would be to run several runs, and take an average of these, but I do not feel that this approach would yield any significant advantage, and it would be much more time consuming.

To make sure that the parameters are not optimized for any particular value of $\sigma$, the tuning process is repeated for $\sigma$ values of 0.0, 0.5 and 1.0 and the optimized parameters are listed in table 6.3.

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>Algorithm</th>
<th>$\alpha$</th>
<th>$\epsilon$</th>
<th>$\lambda$</th>
<th>$\gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>$Q(\lambda)$</td>
<td>0.001</td>
<td>0.001</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>Q-SARSA($\lambda$)</td>
<td>0.01</td>
<td>0.1</td>
<td>0.6</td>
<td>0.8</td>
</tr>
<tr>
<td>1.0</td>
<td>SARSA($\lambda$)</td>
<td>0.001</td>
<td>0.001</td>
<td>0.3</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 6.3: The tuned parameters for a $\sigma$ value of 0.0 which is the $Q(\lambda)$, 0.5 which is an equal part Q($\lambda$) and SARSA($\lambda$) and 1.0 which is the SARSA($\lambda$) algorithm.

Figure 6.3 shows three graphs for the off-line and on-line performance for variations of the $\alpha$ parameter. The on-line performance is an average performance of the first 100,000 episodes and the off-line performance is measured as an average of 100,000 episodes after 10,000,000 episodes of learning. For the three graphs, the $\epsilon$, $\lambda$ and $\gamma$ parameters are fixed at their tuned values for a $\sigma$ value of 0.0, 0.5 and 1.0 respectively. Similarly figure 6.4, 6.6 and 6.5 show graphs for the $\epsilon$, $\lambda$ and $\gamma$ parameters.

For the $\alpha$ parameter in figure 6.3 it can be seen on all three graphs that when the $\alpha$ parameter increases, so does the on-line performance. This is because a higher $\alpha$ value means that the learning will go faster. However, when the learning is faster it is less precise and the off-line performance suffers for this and as the $\alpha$ parameter grows too much, the on-line performance is also influenced by this. The span between the on-line and off-line performance is larger for the graph with parameters tuned for a $\sigma$ value of 0.5. This is because this set of parameters has a higher $\epsilon$ value and with more explorative steps the on-line performance suffers.

The $\epsilon$ parameter in figure 6.4 has a very large influence on the on-line performance, but less influence on the off-line performance. The on-line performance is directly influenced by the fact that more explorative actions are taken, but given that enough episodes are used for training, it does not seem that this variable has much influence on the final off-line result.

When looking at the $\gamma$ and $\lambda$ parameters in figure 6.5 and 6.6, it is interesting to notice that the choice of $\gamma$ and $\lambda$ has close to no influence on the on-line and off-line performance. I believe that the primary reason for this is the fact that the blackjack problem has very short episodes.

The graphs for the variation of the $\alpha$, $\epsilon$, $\lambda$ and $\gamma$ show that although the graphs for the three set of tuned parameters are not identical, it is evident that the $\sigma$ parameter has very little influence on the off-line and on-line results. To verify this observation the three different set of optimized parameters have been tested with different values for $\sigma$. The graphs for the different values of $\sigma$ for the different configurations of parameters can be seen in figure 6.7.

Figure 6.7 clearly shows that for the blackjack problem, the $\sigma$ values has close to no influence on the off-line performance after 10,000,000 episodes. The blackjack problem is not a very difficult problem, and I feel that although Adam White used 10,000,000 episodes to solve the problem, a good reinforcement learning algorithm should be able to find a good policy in much less episodes. For this reason I have measured the off-line performance every 1000 episodes during 100,000 episodes for different $\sigma$ values. The remaining parameters were set to the parameters optimized for a $\sigma$ of 0.5, with the difference that $\alpha$ is set to 0.1, to allow for faster learning.
6.3. **TABULAR Q-SARSA(\(\lambda\))**

![Graph showing variations of the \(\alpha\) parameter for the blackjack problem.](image)

Figure 6.3: Variations of the \(\alpha\) parameter for the blackjack problem, where the remaining parameters are tuned for \(\sigma\) values of 0.0, 0.5 and 0.1 respectively. The average on-line performance is measured after 100,000 episodes, and the off-line performance is measured after 10,000,000 episodes as an average of 100,000 episodes.
Figure 6.4: Variations of the $\epsilon$ parameter for the blackjack problem, where the remaining parameters are tuned for $\sigma$ values of 0.0, 0.5 and 0.1 respectively. The average on-line performance is measured after 100,000 episodes, and the off-line performance is measured after 10,000,000 episodes as an average of 100,000 episodes.
Figure 6.5: Variations of the $\gamma$ parameter for the blackjack problem, where the remaining parameters are tuned for $\sigma$ values of 0.0, 0.5 and 0.1 respectively. The average on-line performance is measured after 100,000 episodes, and the off-line performance is measured after 10,000,000 episodes as an average of 100,000 episodes.
CHAPTER 6. REINFORCEMENT LEARNING TESTS

Figure 6.6: Variations of the $\lambda$ parameter for the blackjack problem, where the remaining parameters are tuned for $\sigma$ values of 0.0, 0.5, and 0.1 respectively. The average on-line performance is measured after 100,000 episodes, and the off-line performance is measured after 10,000,000 episodes as an average of 100,000 episodes.
6.3. **TABULAR Q-SARSA(λ)**

The table and figure illustrate variations of the \( \sigma \) parameter for the blackjack problem, where the remaining parameters are tuned for \( \sigma \) values of 0.0, 0.5 and 0.1 respectively. The average on-line performance is measured after 100,000 episodes, and the off-line performance is measured after 10,000,000 episodes as an average of 100,000 episodes.

![Figure 6.7: Variations of the \( \sigma \) parameter for the blackjack problem, where the remaining parameters are tuned for \( \sigma \) values of 0.0, 0.5 and 0.1 respectively. The average on-line performance is measured after 100,000 episodes, and the off-line performance is measured after 10,000,000 episodes as an average of 100,000 episodes.](image-url)
Figure 6.8 clearly shows that a good policy can be found within the first 100,000 episodes, although it is not as good as the policy found after 10,000,000 episodes. The figure also shows that for the blackjack problem the $\sigma$ parameter is of little importance even in the initial phase, which also means that SARSA($\lambda$) and $Q(\lambda)$ performs equally well for this problem.

The $Q$-SARSA($\lambda$) algorithm has given good results for the blackjack problem, but the results were no better than the results that could be received by the $Q(\lambda)$ or the SARSA($\lambda$) algorithms alone. This could be seen as an indication that the $Q$-SARSA($\lambda$) is no better than the other two algorithms, but I do not believe that this is the case. It is known that for some problems there is a difference in the learning curve for $Q(\lambda)$, and SARSA($\lambda$), which is a clear indication that there will also be a difference in the learning curve for different values of the $\sigma$ parameter. However, for the blackjack problem this was not the case, so the only clear results that could be gathered from this test, is the fact that for some problems the $Q$-SARSA($\lambda$) algorithm will be no better than the $Q(\lambda)$ or the SARSA($\lambda$) algorithms. The effect of the $\sigma$ parameter, will be further evaluated for the more advanced mountain car, cart pole and backgammon problems, when they will be learned by the incrementally, batch, cascading and NFQ-SARSA($\lambda$) trained neural networks, so although the blackjack problem can not produce any conclusive results, there is still a chance that other problems will produce more conclusive results.

### 6.4 On-line Incremental Neural $Q$-SARSA

As described in section 5.7 on page 88, replacing eligibility traces have been implemented in the sliding window cache, and is for this reason not supported for the incrementally trained neural networks, so only $Q$-SARSA is supported, and not $Q$-SARSA($\lambda$). This is as such not a problem, since the primary focus of this test is on batch training and not on incremental training. However, I will make some tests on the mountain car and cart pole problems, which can be used as a basis for
6.4. ON-LINE INCREMENTAL NEURAL Q-SARSA

comparison. Both problems use 10,000 episodes for learning, with 1000 episodes for measuring the off-line performance afterwards, and both problems use the same neural network parameters as was used for the the neural network benchmarks. Like for the tabular test, the random number generator is also seeded with a fixed value for these tests, to allow for the tests to be reproducible.

6.4.1 Incremental Q-SARSA for Mountain Car

The mountain car problem has two continuous states and 3 discrete actions. The actions can either be represented as a single input neuron, with +1 for full forward, -1 for full reverse, and 0 for no throttle, or it can be represented as 3 individual neurons. With only three actions, it would probably be easier for the ANN to generalize with the actions represented as individual neurons, and this was also what the preliminary tests showed, but the difference between the two representations did not seem to be very significant. The final neural network ended up with 5 input neurons, one for each of the state variables and one for each action. The hidden layer included 20 hidden neurons with a symmetric sinus activation function.

With only 300 steps to reach the goal as dictated by the NIPS 2005 workshop rules, many of the episodes were ended before the agent found the goal. This posed a problem for learning, and especially agents with low values for $\alpha$ had difficulties learning anything useful. Figure 6.9 clearly shows that low value for $\alpha$ leads to low on-line performance. If the on-line performance is too low, the off-line performance will also be very low, because there has not been enough successful runs during learning to learn a successful policy.

![Graph showing on-line and off-line performance for Mountain Car](image)

Figure 6.9: On-line and off-line performance, for the incrementally trained mountain car problem, for variations of the $\alpha$, $\epsilon$, $\gamma$ and $\sigma$ parameter, with the remaining parameters fixed at their tuned values. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.

Figure 6.9 shows that for the mountain car problem, the $\sigma$ parameter is of great...
importance and only a $\sigma$ value close to zero will yield any significant learning. A $\sigma$ value close to zero is a $Q$-SARSA algorithm close to $Q$-learning, so for some reason $Q$-learning is superior to SARSA learning for this problem. It is hard to say why this is the case and as will be evident when the other algorithms have been tested, the same $\sigma$ is not optimal for all algorithms used on the mountain car problem. This phenomenon is discussed further in section 6.9.1 on page 151.

A policy which used 165 steps on average during the 1000 off-line episodes, was learned with parameters fixed at $\alpha = 0.2$, $\epsilon = 0.2$, $\gamma = 1.0$ and $\sigma = 0.0$. When this simple model-free algorithm is compared to the algorithms from the NIPS conference it performs surprisingly well. The algorithms at NIPS learned policies that were able to reach the goal within anything from 75 to 120 steps, which of course is far better than 165 steps, but the gap is not nearly as large as could have been expected.

### 6.4.2 Incremental $Q$-SARSA for Cart Pole

The cart pole problem has 21 actions, and these could be represented as separate inputs for the ANN, just like the actions for the mountain car problem. However, the actions are ordered in a way, where action values that are close to each other share properties, so in order to help generalization the actions are represented as one input to the ANN. In combination with the four states, this leads to an ANN with 5 inputs, and after a number of trials with different number of hidden layers and neurons, a network structure with one hidden layer containing 20 neurons with the symmetric sinus activation function was selected.

Even with the modified reward structure, the cart pole problem is difficult to solve for a simple incrementally trained reinforcement learning algorithm, and many combinations of parameters failed to produce a policy that was able to balance the pole for a longer period. However, with tuning of the parameters, it was possible to produce a policy that was able to keep the pole from falling within the 300 steps in 800 of the 1000 off-line episodes. This policy had an average reward of 68 per episode, and an average number of steps of 262. It is difficult to compare these results to that of the NIPS 2005 conference since a different reward structure is used here. However, since there is nothing that prevented the contestants at the conference from using a different reward structure internally, it is still possible to compare the average number of steps. The majority of the algorithms was able to balance the pole for 200-300 steps, while two algorithms was unable to balance the pole and two managed to balance the pole for the complete 300 steps. This means that the $Q$-SARSA algorithm actually performs comparable to the algorithms at the NIPS conference, which is a surprise.

The parameters that was used to produce this policy was $\gamma = 0.5$, $\alpha = 0.01$, $\epsilon = 0.1$ and $\sigma = 0.1$, and the on-line and off-line performance for variations of the $\sigma$ value can be seen in figure 6.10.

Figure 6.10 shows that especially the off-line reward is very jumpy, and although the graph indicates that 0.1 is the optimal $\sigma$ parameter it might just be a coincidence. In order to investigate this phenomenon further, ten runs was generated with different random seeds, but otherwise with the same parameters as in figure 6.10. The minimum, maximum and average performance for these ten runs can be seen in figure 6.11. The on-line performance has not changed much, but the off-line performance can be seen to be very unpredictable, and for all values of $\sigma$ there is a chance of getting a performance close to -1000, meaning that the pole will fall immediately. However, there is also a good chance of learning a good policy that will be able to balance the pole for more than 200 steps on average. Although there does seem to be an indication that a higher $\sigma$ value will give better results, the results are not conclusive.
There may be many reasons why the off-line performance for incremental $Q$-SARSA is so unpredictable for the cart pole problem, when the on-line performance is not. I believe that the most important factor is the last few episodes before the learning is ended. Incremental training is very biased towards the last seen training patterns, so the state of the ANN will very much depend on the last few episodes. This means that the final learned policy will also be very dependent on the last few episodes. If valuable information has been learned during the last few episodes, the performance of the resulting policy will be good, otherwise the performance will not be as good.

The incrementally trained $Q$-SARSA algorithm is able to produce relatively good results for the mountain car and cart pole problems, and the results serve as a good baseline for evaluating the batch and cascading $Q$-SARSA($\lambda$) algorithms.

### 6.5 Batch Neural $Q$-SARSA($\lambda$)

The batch $Q$-SARSA($\lambda$) algorithm will be able to take full advantage of the sliding window cache (see section 5.4 on page 79), and will be able to include replacing eligibility traces. The algorithm will use iRPROP$^-$ as the neural network training algorithm, since it was the best performing non-cascading algorithm in the neural network benchmarks. iRPROP$^-$ generally performs better than incremental back-propagation for neural network training, but it must be kept in mind that reinforcement learning is an incremental problem, which gives the incremental algorithm a definite advantage. The sliding window cache is designed to minimize the disadvantages of using a batch algorithm, and should give the batch $Q$-SARSA($\lambda$) algorithm an advantage compared to the incremental algorithm.

This section will determine whether the sliding window cache in combination with iRPROP$^-$ is able to beat the performance of the incrementally trained neural network. In order to make this comparison as bias free as possible, the same input
Figure 6.11: The minimum, maximum and average on-line and off-line performance, for the incrementally trained cart pole problem. Measured during ten individual runs, as a function of the \( \sigma \) parameter, with the remaining parameters fixed at their tuned values. The graph represent the average performance, while the vertical bars represent the span between the minimum and maximum performance. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.
representation and neural network topology will be used in this test as was used in incremental test.

6.5.1 Batch $Q$-SARSA($\lambda$) for Mountain Car

With the sliding window cache and replacing eligibility traces, three new parameters have been added to the reinforcement learning algorithm. The new parameters are the $\lambda$ parameter for the eligibility trace, and for the sliding window cache the cache size and cache commit interval have been included. The cache commit interval determines how often the data from the sliding window cache should be used to train the neural network. The interval is measured in cache writes, and since one $Q(s, a)$ value is written to the cache for each step (when rehearsing is not used), the interval also represents the number of steps between each training epoch.

These extra parameters give more opportunities, but also makes it more difficult to find the correct configuration of parameters. This is especially a problem for the mountain car problem, because many episodes will end without finding the goal, and if the parameters are not tuned correctly, a policy that is able to find the goal within 300 steps will never be found. In order to find appropriate parameters, the parameters are tuned as described in section 6.2. This was the same approach as was used in the tabular and incremental tests, and the risk of tuning the parameters to reach a local minimum instead of the global minimum is an even greater issue in this case, because there are more parameters to tune. The tuned parameters are $\gamma = 0.5$, $\lambda = 0.9$, $\alpha = 0.1$, $\epsilon = 0.2$, $\sigma = 1.0$, a cache size of 500 and a cache commit interval of 10 steps. This configuration learned a policy which on average was able to reach the goal in 155 steps in the 1000 off-line episodes. This is better than the 165 steps which was reached using the incremental $Q$-SARSA algorithm, and this should as such suggest that the batch $Q$-SARSA($\lambda$) algorithm is better than the incremental algorithm for the mountain car problem. However, with such a small margin between the two values, a closer look at the individual parameters will be needed in order to provide a clearer picture. Figure 6.12 shows a closer look at the on-line and off-line performance for different values of $\epsilon$, $\alpha$, $\sigma$ and $\gamma$, with the remaining parameters fixed at their tuned values, similarly variations for $\lambda$ can be seen in figure 6.13, and variations for the cache size and the cache commit interval can be seen in figure 6.14.

The most significant difference between the off-line and on-line performance for the $\epsilon$ parameter in figure 6.12 and the incrementally trained performance in figure 6.9 on page 107, is that the batch algorithm performs worse for low values of $\epsilon$. This indicates that the batch algorithm needs more exploration than the incremental algorithm, in order to produce a good policy. This is not that surprising, considering that incremental back-propagation has a built in stochasticity, since it in every step adjusts the network weights by only looking at one training sample. This stochasticity, is the same stochasticity which makes the incremental back-propagation escape a local minimum as discussed in section 2.3.1 on page 14. When a higher level of exploration is used, the batch algorithm outperforms the incremental algorithm, although the margin is not large enough to give any conclusive results.

While only large values for $\alpha$ was able to learn a good policy for the incrementally trained mountain car problem, the same values are not able to learn a good policy for the batch trained problem. The batch $Q$-SARSA($\lambda$) algorithm show far better results for the low $\alpha$ values, which I believe is a direct effect of the extra neural network training that the cache introduces. When the cache introduces more training, the $Q$-SARSA($\lambda$) update rule need not include as high a learning rate. However, the difference in requirements for the $\alpha$ parameter may also be influenced by the eligibility trace, which is included in the batch algorithm. The eligibility
traces speeds up propagating the goal reward back to the other states of the environment, and this may explain why a high $\alpha$ value is not needed, but it does not explain why a high $\alpha$ value is not able to produce a good policy. For this reason I believe that the extra training provided by the cache has more to do with the difference, than the $\lambda$ parameter.

The optimal $\sigma$ value for the batch $Q$-SARSA($\lambda$) is 1.0, which is the SARSA($\lambda$) algorithm. This is contrary to the optimal value for the incremental algorithm which was 0.0 and the $Q(\lambda)$ algorithm. The batch algorithm is able to learn reasonable good policies for a broad distribution of $\sigma$ values ranging from 0.0 to 1.0, which is in striking contrast to the incremental algorithm which was only able to learn anything useful for $\sigma \leq 0.2$. It is difficult to say why the $\sigma$ parameter is of greater importance for incremental training than for batch training. A theory could be that the sliding window cache ensures a more global view, so the cost of exploration is not that important to the overall learned policy, but it is not possible to say for sure if this is the case and this issue will be discussed further in section 6.9.1 on page 151.

The incremental $Q$-SARSA was very tolerant to different values of the $\gamma$ parameter, and could produce good policies for 7 out of 11 possible values. The batch $Q$-SARSA($\lambda$) algorithm was only able to produce good policies for two of the 11 values, and although the best policy for the batch algorithm is better than the best policy for the iterative algorithm, the results for the $\gamma$ parameter is not encouraging. It is hard to say why the $\gamma$ parameter is more important for the batch algorithm, than it is for the incremental algorithm, but it might have something to do with the fact that the $\gamma$ parameter is also a large factor in the eligibility trace, which will be discussed later in this section, when the $\lambda$ parameter in figure 6.13 is discussed.
Generally for \( \alpha \) and \( \sigma \), the batch algorithm is able to learn a good policy for more combinations of parameters, while the incremental algorithm is able to learn good policies for more combinations of the \( \epsilon \) and \( \gamma \) parameter. This places the two algorithms very close, with a slight upper hand to the batch algorithm, since it produced the best policy. However, this policy was produced by using eligibility traces which was not available to the incremental algorithm.

The \( \lambda \) parameter in figure 6.13 was not included in the incremental training because the incremental training did not include eligibility traces. It is interesting to see that for this parameter, close to no learning occurs when \( \lambda \) is zero. The parameters are not optimized for a \( \lambda \) parameter of zero, so it is hard to say how good the learning would be if batch learning did not include eligibility traces. However, I do not suspect that the learning would be as good as for the current parameters, and I doubt that a batch algorithm with \( \lambda \) set to zero would be able to produce a policy that could compete with the policy learned through incremental training. This indicates that a reason for the batch learning being more successful than incremental learning might be that it includes replacing eligibility traces.

A new dimension, which was not included in the incremental algorithm is the parameters for the sliding window cache. The cache size and cache commit interval which is displayed in figure 6.14 shows a clear picture of the performance, with a good correlation between the on-line and off-line performance. The optimal cache size is 500, and the optimal cache commit interval for this cache size is 10, leading to all elements in the cache being used for training 50 times before they leave the cache. Training less frequently than every 10 steps will lead to worse performance, which is not that surprising, but it is surprising that more frequent training also weakens performance. I believe that the reason for this, is that more frequent training leads to over-fitting. Over-fitting is a very undesirable situation, when the function that should be approximated is constantly changing. Early over-fitting may lead to a situation, where the neural network gets stuck and is unable to recover.

The optimal cache size is 500, which does not seem that large, since many
6.5.2 Batch $Q$-SARSA($\lambda$) for Cart Pole

The batch $Q$-SARSA($\lambda$) algorithm had many problems learning a good policy for the cart pole problem, however, one particular configuration of parameters proved to give surprisingly good results for the problem. The configuration managed to achieve an average number of off-line steps of 297 and an off-line reward of 311. The average number of off-line steps indicates that the policy was able to balance the pole for the full 300 episodes in almost all of the episodes, and the average reward of 311 indicates that the pole was completely balanced in many of the steps.
The configuration which achieved this performance had the parameters fixed at \( \alpha = 0.01, \epsilon = 0.001, \sigma = 0.4, \gamma = 0.9, \lambda = 0.9, \) a cache size of 2000 and a cache commit interval of 5.

It would be tempting to simply conclude that the batch algorithm is far superior to the incremental algorithm, but when the graph for variations of the \( \sigma \) parameter in figure 6.15 is compared to figure 6.10 which shows the same graph for the incremental algorithm, it shows that this conclusion can not be made as simple.

![Figure 6.15: On-line and off-line performance, for the batch trained cart pole problem, for as a function of the \( \sigma \) parameter, with the remaining parameters fixed at their tuned values. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.](image)

The graph for \( \sigma \) for the incremental algorithm shows that several different values of \( \sigma \) is able to produce a policy that exhibits some level of learning. However, the same graph for the batch algorithm shows only that a \( \sigma \) parameter of 0.4 is able to learn anything useful. There does not seem to be any logical reason as to why a \( \sigma \) value of 0.4 should be significantly better than a \( \sigma \) value of 0.3 or 0.5, so a closer look is needed before any conclusion can be made.

The on-line cumulative reward is often used as a way of measuring the performance of a reinforcement learning algorithm, and although I feel that this method punishes exploration too much, it does show a very clear picture of how the learning evolves. Figure 6.16 shows the cumulative reward for variations of the \( \sigma \) parameters, and shows exactly how the learning evolves differently for a \( \sigma \) value of 0.4.

For the first 7000 episodes, all of the different \( \sigma \) values perform equally well, but then suddenly it seems like the \( \sigma \) value of 0.4 stumbles upon a very good policy, and from one episode to the other it performs far better than all the other values. Figure 6.17 shows a closer look at the number of steps per episode and the average mean square error (MSE) per episode during episode 6000 to episode 8000, and can shed light on what exactly happened during these 2000 episodes.

Figure 6.17 shows that the learning does not just stumble upon a good policy. After 6650 episodes the learning experiences a few episodes where the pole is being balanced for more than 50 steps. This is not unusual, and the same has happened for all of the other values of \( \sigma \). What is unusual in this situation is the fact that the \( Q(s,a) \) values learned during these episodes appears to be significantly different.
Figure 6.16: The cumulative on-line reward, for the batch trained cart pole problem for variations of the $\sigma$ parameter with the remaining parameters fixed at their tuned values.

Figure 6.17: The number of steps per episode and the average MSE per episode for the batch trained cart pole problem, for the tuned parameters and a $\sigma$ value of 0.4 during episode 6000 to episode 8000.
from the previously learned values, which seems to indicate that a new strategy appears at this point. The new strategy makes the MSE increase dramatically, and during the next hundred episodes, the pole is not being balanced for more than 10 to 20 steps, but the MSE still increases, which seems to indicate that the newly learned strategy still has some effect on the $Q(s, a)$ values. After about 100 episodes the new strategy is finally learned, and during the next 20 episodes, the MSE decreases quickly while the number of steps quickly increases. At this point, the learning goes into a convergence phase, where the MSE slowly decreases, while the number of steps per episode slowly increases. After about 450 episodes in this phase, the MSE finally converges to a value below 1.0, and the number of steps per episode reaches 300. From this point on, the new strategy seems to be learned and it is followed, which is the effect that can be seen in the graph of the cumulative reward.

The sliding window cache played a central role in the learning of the new strategy, and I do not believe that the policy could have been learned without the cache. The cache had a size of 2000, which indicates that during the 100 episodes between the first spike in the number of steps per episode and the second spike, the experience from the first spike was still in the cache. During this period, the first spike had significant impact on what the neural network learned, and a close look at the number of steps per episode revealed that the first spike was still in the cache when the second spike happened, but the spike was at the very end of the cache. This indicates, that the second spike happened because the $Q(s, a)$ values learned before the first spike finally left the cache, and the new more effective strategy could take over from the old. If there had not been a cache, I do not believe that the first spike would be able to have enough influence on the policy, to change it quite as dramatically as seen in this example.

Although the sliding window cache was one of the main contributors to the learned policy, this does not eliminate the fact that figure 6.15 suggests that it was a coincidence that such a good policy was learned. The good policy was reached with the random number generator seeded at a fixed value, which means that it is reproducible, but other random seeds need be tested to see if it in fact was a coincidence that the policy was found. Figure 6.18 shows the result of running the learning with ten different random seeds, for different values of the $\sigma$ parameter.

Figure 6.18 shows that a $\sigma$ value of 0.4 is generally not any better than the other values. If the figure is compared to the results for the incremental learning in figure 6.11 on page 110 it is clear to see that generally the incremental algorithm performs better than the batch algorithm for this particular problem.

Although it was possible to produce a better policy, for the batch algorithm than the incremental algorithm for both the mountain car and the cart pole problems, it is not possible to conclusive say which algorithm is the better. However, the tests have clearly showed that the sliding window cache is able to successfully combine advanced neural network training algorithms with reinforcement learning. The following section will determine how the incremental and batch algorithm compares to the cascading algorithm.

### 6.6 Cascading Neural $Q$-SARSA($\lambda$)

The cascading neural $Q$-SARSA($\lambda$), using the “Cascade2 RPROP Multi” configuration from section 3.3, should be able to adapt more easily to a problem than the incremental and batch algorithms. One particular feature of the algorithm that should be especially beneficial is the fact that the neural network starts out without any hidden neurons. A neural network without hidden neurons can not learn any advanced functions, but it is extremely fast at learning simple linear functions. This ability should enable the algorithm to quickly develop a simple strategy for tackling
Figure 6.18: The minimum, maximum and average on-line and off-line performance for the batch trained cart pole problem. Measured during ten individual runs, as a function of the $\sigma$ parameter, with the remaining parameters fixed at their tuned values. The graph represent the average performance, while the vertical bars represent the span between the minimum and maximum performance. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.
the problem. The simple strategy will not perform very well, but it will be learned much faster than any strategy that is learned by a multilayered network. The simple strategy will be combined with more advanced strategies as more neurons are added to the network, and will ensure that the policy will evolve over time to a more advanced policy. However, the cascading algorithm has two issues that may hinder its performance:

**The growth of the network:** If the network grows too fast, many neurons will be added to the network that does not contribute to the quality of the network, and learning will be slowed down considerably. If the network grows too slow not enough neurons will be installed, to enable approximation of an advanced $Q(s, a)$ function.

**The training of the candidates:** The candidates are trained very intensely with the data patterns that are available in the cache. This means that the patterns that are in the cache when the new candidates are being trained is of great importance, while training patterns that are in the cache when the candidates are not trained are not equally important.

The experiments in this section will show how the candidate training compares to the incremental and batch algorithms, and it will show exactly how important the two issues are for the final performance.

### 6.6.1 Cascading $Q$-SARSA($\lambda$) for Mountain Car

The incremental $Q$-SARSA and the batch $Q$-SARSA($\lambda$) algorithms have learned policies capable of reaching the goal in 165 and 155 steps. This is not directly comparable to the results from the NIPS 2005 conference of 75 to 120 steps, but this was not expected and neither can it be expected for the cascading $Q$-SARSA($\lambda$) to perform comparable to these results.

The cascading neural $Q$-SARSA($\lambda$) has an advantage compared to the two other algorithms, in the fact that the neural network training algorithm is more effective in the neural network benchmarks, and because the evolving nature of the network should be able adapt a simple policy at first, and evolve that policy into a better policy. However, the mountain car problem has two issues, which makes it very hard for the neural network to learn anything useful in the first few episodes, before any candidate neurons are added to the network. The first problem lies in the fact that it is very hard to actually reach the goal within 300 episodes, so many of the initial episodes will have a very hard time learning anything useful. The second problem is the fact that no simple linear function is able to represent a policy that will find the goal reliable within 300 episodes. The policy needs to determine the correct action based on a non-trivial combination of position and speed, and there is no guarantee that positions and speeds close to each other will require the same action. However, figure 6.19 clearly shows, that the cascade algorithm is able to produce a better policy than the incremental and batch algorithm.

The best performing policy is able to reach the goal in an average of 146 steps during the 1000 off-line episodes, with tuned parameters of $\alpha = 0.2$, $\epsilon = 0.01$, $\gamma = 0.6$, $\sigma = 0.4$, $\lambda = 0.3$, a cache size of 500 and a cache commit interval of 20. 146 steps is quite good compared to the 155 steps by the batch algorithm and the 165 steps by the incremental algorithm, and it is getting closer to the 75 to 120 steps which was reached by the algorithms at the NIPS conference.

The graphs for the $\alpha$, $\epsilon$, $\gamma$ and $\lambda$ parameters in figure 6.19 does not differ too much from the same graphs for the incremental and batch trained algorithms, except for the fact that it seems like the on-line performance is generally better for the cascading algorithm. This is consistent with the theory, that the cascading
neural Q-SARSA($\lambda$) should be able to adapt more quickly to a good policy, but it may just as well be a consequence of the $\epsilon$ parameter, which is tuned to a smaller value for the cascade algorithm which means that the exploration penalty will be smaller.

Figure 6.20 shows the cumulative reward for the batch and cascade Q-SARSA($\lambda$) algorithms, for variations of the $\gamma$ parameter. Three observations can be made from the graphs in figure 6.20: The first being that the cumulative reward is significantly better for the cascading algorithm than for the batch algorithm. The second is that the cascade algorithm learns more quickly, hence making the curve for the cumulative reward break earlier. The third observation is that the learning is more jerky, and seems to happen more in steps. The first observation is the same as was made on the graphs for the on-line and off-line performance, and that can easily be contributed to the $\epsilon$ parameter. The second and third observations could also be contributed to the $\epsilon$ parameter, but they are more consistent with the theory, that a cascading Q-SARSA($\lambda$) algorithm will quickly adapt a simple but good policy and then gradually evolve better and better strategies as more candidate neurons are added.

Figure 6.21 shows the variations for the cache size and the cache commit interval for the cascade algorithm. The cascade algorithms only has a few values for the cache size and the cache commit interval that will give good off-line results, which was also the case for the batch algorithm. However, the main difference is that for the cascade algorithm, the difference between the on-line results are not very large, and fairly good on-line results are received for all variations of cache size and cache commit interval.

This observation is not that surprising since the cascade algorithm generally
Figure 6.20: On-line cumulative reward for the batch (top) and cascade (bottom) $Q$-SARSA(\(\lambda\)) algorithms, for the mountain car problem, for variations of the $\gamma$ parameter.
provided better on-line results for the $\alpha$, $\epsilon$, $\gamma$ and $\lambda$ parameters. A closer look at what happens during the training does, however, reveal something interesting, which makes the good on-line performance seem more surprising. Figure 6.22 shows how the candidates are added during the 10,000 episodes, and it can be seen that although the candidates are added at a moderate speed for the tuned parameters, ending with 31 candidates, for some cache sizes and cache commit intervals, the candidate neurons are added much faster.

When candidate neurons are added too quickly, they will not contribute much value to the final policy and further learning will be difficult, this means that the learning will seriously be hindered and a good policy can not be expected. Another aspect of the addition of new candidates, is the fact that training candidates and adding them to the network is a time consuming task. In order to be able to run all the benchmarks, I was forced to stop the learning after 2 hours, which meant that e.g. the commit interval of 1 was stopped after only a few hundred episodes. I do not suspect that any of these agents would be able to produce a good policy, since they added neurons far too quickly, so I do not view the early stopping as a problem. However, a very interesting aspect of these agents, is the fact that even though they add candidate neurons at a rate where not much learning should be expected, they still manages to keep a fairly high on-line reward. It is hard to say exactly why the on-line reward is so high, but it seems to suggest that the cascading algorithm is a bit more robust when many neurons are added, than initially thought. However, none of these agents were able to learn a good off-line policy, so they were still hindered in their learning.

Only variations of the cache size and the cache commit interval had problems with neurons being added too quickly. This is not that surprising, since they are the only parameters which directly influence the cache. Section 8.1 on page 159 takes a closer look at this influence, and discussed which consequences that this influence has on the learning.

Section 5.6 on page 86 discussed the whole ANN method of making sure that the neural network does not grow too rapidly, and to make sure that the inserted candidate neurons are used optimally. This method is described in algorithm 7 and trains the entire network in between only training the output neurons. This method was not used for the mountain car and cart pole problems, because tests showed that the whole ANN training did not perform as well as the traditional cascade training, where only the outputs are trained. Figure 6.23 shows a comparison between the
6.6. CASCADING NEURAL Q-SARSA(λ)

Figure 6.22: The number of candidates over time, for the cascade trained mountain car problem, for different cache sizes and commit intervals. Some of the graphs are ended before the 10,000 episodes, because the training is ended after a fixed period of 2 hours.
normal approach, and the whole ANN approach, for variations of the $\sigma$ parameter.

Figure 6.23: On-line and off-line performance, for the cascade trained mountain car problem, for variations of the $\sigma$ parameter, with the remaining parameters fixed at their tuned values. The left graph has included the modification to the cascade architecture suggested in algorithm 7 on page 87, where the whole ANN is trained after the outputs have been trained. The right graph does not include this modification. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.

The tendency in figure 6.23 was reproducible for other parameters, and the tendency was clear. Whole ANN training generally performed worse, but not with a wide margin, so perhaps there is some use for the method.

Similarly the method of rehearsing, which was discussed in section 5.3 on page 78 was tested on the mountain car and cart pole problems. Rehearsing quickly showed that it dramatically hindered the learning, and I believe that this is mostly due to the fact that full rehearsing is the only implemented rehearsing strategy. Full rehearsing trains using all possible actions, while Rivest and Precup (2003) suggested that it might be an idea to only rehearse using a few other actions than the taken action. Other rehearsing strategies was not tested, but it could be and idea as a future project to investigate various ways of implementing rehearsing.

6.6.2 Cascading $Q$-SARSA($\lambda$) for Mountain Car Revisited

As discussed in the earlier section, the results from this implementation is not able to compare to the results from NIPS 2005. This was not expected, since many of the algorithms at the NIPS conference was highly optimized algorithms, and because they are able to use knowledge about the structure of the mountain car problem to increase their performance.

For this thesis it is not desirable to include any problem specific tuning, but perhaps the performance of the cascading $Q$-SARSA($\lambda$) can be enhanced by other means. It was originally thought that handling the three actions as separate inputs would be more beneficial than handling them as one input, and some of the initial experiments supported this belief. However, it could be an idea to see how the $Q$-SARSA($\lambda$) algorithm would perform if the actions was handled as one input. A tuning session was made with the representation, where the action was only represented as one input, and as shown in figure 6.24 it proved to produce better results.

When the action was only represented as one input, the cascade $Q$-SARSA($\lambda$) algorithm was able to produce a policy that could reach the goal in an average of 111 steps in the 1000 off-line episodes. This was reached with parameters fixed at:
6.6. CASCADING NEURAL Q-SARSA(\(\lambda\))

Figure 6.24: On-line and off-line performance, for the cascade trained mountain car problem, for variations of the \(\alpha\) parameter, with the remaining parameters fixed at their tuned values. The left graph only represents the action as one input, while the right implements it as three. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.

\[\alpha = 0.5, \epsilon = 0.2, \gamma = 0.6, \lambda = 0.6, \sigma = 0.4, \text{ a cache size of 500 and a cache commit interval of 20}.\]

111 steps is not comparable to the best results from the NIPS conference, but it does compare to several of the results from the conference. This is very promising for the cascading Q-SARSA(\(\lambda\)) algorithm, since it was not expected that Q-SARSA(\(\lambda\)) could compete with the optimized algorithms for smaller problems.

The explanation as to why this representation is better must lie in the fact that there is a clear relationship between the three actions, where it is often the case that; if full forward throttle is the best action, then no throttle is the second best and reverse is the worst. When this representation is able to produce better results for the cascade Q-SARSA(\(\lambda\)) algorithm, then it will probably also be able to enhance on the performance of the batch Q-SARSA(\(\lambda\)) and the incremental Q-SARSA algorithms. However, there is no reason to test this in greater detail, since the purpose of these benchmarks are primarily to test the three configurations against each other, and this has already been tested using actions represented as three inputs.

6.6.3 Cascading Q-SARSA(\(\lambda\)) for Cart Pole

For the incremental Q-SARSA and the batch Q-SARSA(\(\lambda\)) good policies have been learned, but experiments with random seeds showed that the results were much dependent on the random seed. The cascade algorithm should have an advantage, because it is able to easy find a simple policy that can balance the pole for a couple of steps, and subsequent candidate neurons will be able to enhance this policy.

Using the Q-SARSA(\(\lambda\)) algorithm, it was possible to find a good policy, and several configurations of parameters was able to produce a policy that was able to balance the pole for all of the 300 steps in all of the 1000 off-line episodes. One such combination is the combination where the parameters are fixed at \(\alpha = 0.0001, \epsilon = 0.01, \gamma = 0.9, \lambda = 0.9, \sigma = 0.4, \text{ a cache size of 2000 and a cache commit interval of 5}\). With this combination, the pole was balanced for all of the 300 steps in all of the 1000 off-line episodes, and the average off-line reward was 318. A closer look at the different values for the \(\sigma\) parameter revealed that a \(\sigma\) parameter of 0.5 would give an average off-line reward of 326, so although the tuned parameters do provide a policy that is able to balance the pole for all of the 300 steps, the parameters could be tuned even further to provide the best possible off-line reward. These
results are directly comparable with the best results from the NIPS conference and the fact that a broad range of parameters is able to learn such a good policy is very satisfying.

The benchmarks for the cart pole problem using the incremental and batch algorithms have mostly focussed on the fact that it was not possible to learn a good policy for a wide selection of the $\sigma$ parameter. Using cascading $Q$-SARSA($\lambda$) algorithm, this is, however, not a problem, so these benchmarks will focus on the individual parameters, and investigate why it is possible to find a good policy. Figure 6.25 shows the average number of steps for variation of the $\alpha$, $\epsilon$, $\gamma$, $\lambda$ and $\sigma$ parameters. In this figure it is interesting to see that several different values of the $\epsilon$ and $\sigma$ parameters are able to produce policies that are able to balance the pole for all the 300 episodes.

It is very encouraging that the cascade $Q$-SARSA($\lambda$) algorithm is able to learn
such good policies for that many different parameters. However, looking at the parameters themselves does not provide much information about why the algorithm performs so well, but the high on-line performance suggests that the good policies are learned early. Looking at the cache size and cache commit interval, reveals some more information. A cache size of 2000 and a cache commit interval of 5, are parameters which would make the mountain car problem add candidate neurons at an alarming rate and which would not allow much learning to happen. For some reason this is not a problem for the cart pole problem, and when looking in figure 6.26 it is clear to see that a smaller cache size or a larger cache commit interval does not produce good policies.

Figure 6.26: Average number of steps in the on-line and off-line case, for the cascade trained cart pole problem, for variations of the cache size and the cache commit interval, with the remaining parameters fixed at their tuned values. The on-line and off-line steps is measured after 10,000 episodes, where the off-line steps is an average of 1000 episodes.

The large cache size and small cache commit interval should produce a neural network with many hidden neurons, but when looking in figure 6.27 it is interesting to see that this is far from true. After around 300 episodes one candidate neuron is added to the network, and after this no more neurons are added. The reason that no more neurons are added is that when the MSE is under 0.0001 it is deemed that there is no reason for adding new neurons. A close look at the MSE reveals that when the candidate is added to the network after 295 episodes the MSE drops from 0.001566 to 0.000647, and 3 episodes after the candidate have been added to the network, the MSE is below 0.0001. The episode where the candidate is added, is the first episode where the pole is balanced for 300 steps, and it is also the first episode where the pole is balanced for more than 100 steps. After the candidate has been added a couple of episodes is used to stabilize the MSE, and for the next 3 episodes only 1 episode is able to balance the pole for the 300 steps. After this short stabilization phase, there are only 5 of the remaining 9702 episodes, where the pole is not balanced for the complete 300 steps, and in all of these 5 episodes, the pole is balanced for more than 250 steps.

The sharp drop in MSE is caused by the fact that the candidate neuron which is added to the network, is trained to exactly fit the need of the network. I do not believe that it would be possible to do this with a smaller cache, and for this reason I believe that the large cache is central to the success of this algorithm.

The cart pole problem really showed some of the potential that the cascading Q-SARSA(λ) algorithm has, and the results are just as good as the best of the results from the NIPS 2005 conference.

The cart pole problem is an avoidance control problem, where the valid region is very small, but even though it is difficult to find a policy that will keep the
Figure 6.27: The number of candidates over time, for the cascade trained cart pole problem, for different cache sizes and commit intervals. Some of the graphs are ended before the 10,000 episodes, because the training is ended after a fixed period of 2 hours.
pole balanced, the actual learned policy is not that complicated, since the final neural network only contains one hidden neuron. The results here clearly show the potential of the cascading $Q$-SARSA($\lambda$) algorithm, but it still remains to be shown whether the algorithm can scale to larger problem sizes.

6.7 Neural Fitted $Q$-SARSA($\lambda$)

Section 5.5.2 on page 84 suggests that the NFQ-SARSA($\lambda$) algorithm might perform better than the $Q$-SARSA($\lambda$) algorithm. This section will explore if NFQ-SARSA($\lambda$) is able to perform better than the $Q$-SARSA($\lambda$) algorithm. The results for the cascading neural $Q$-SARSA($\lambda$) algorithm has shown that the cascading algorithm generally performs better than the batch and incremental algorithms, so for this reason only the cascading algorithms will be tested in this section.

NFQ was one of the algorithms that was represented at the NIPS workshop, and the results are documented in Dutech et al. (2005). The results show that a good policy can be found for the mountain car and cart pole problems after only a few episodes. A policy which is able to reach the goal in less than 90 steps was found for the mountain car problem in less than 60 episodes, and a policy which was able to balance the pole for 300 steps was found in less than 10 episodes. The details about the NFQ implementation used in Dutech et al. (2005) is very sparse, but Riedmiller (2005) suggests that NFQ is combined with heuristics for adding more patterns from the goal region to the learning set. Riedmiller (2005) do not provide much information about how early and how often the neural network is trained using NFQ, but it seems that the cache is used for training very early and that the cache is also used several times for training between steps. NFQ-SARSA($\lambda$) is not used for training before the cache is full, and then the network is only trained for one epoch after each cache commit interval. For these reasons NFQ-SARSA($\lambda$) can not be expected to learn a good policy quite as fast as the NFQ at the NIPS workshop, but since the NFQ-SARSA($\lambda$) algorithm should use the gathered knowledge more efficiently than the $Q$-SARSA($\lambda$) algorithm, and since the algorithm is far more time consuming, the algorithm is only given 1000 learning episodes as oppose to the 10,000 episodes for the $Q$-SARSA($\lambda$) algorithm.

The cascading NFQ-SARSA($\lambda$) algorithm will be tested on the mountain car and cart pole problems in the next sections, with 1000 on-line and 1000 off-line episodes. The parameters are the same as for the $Q$-SARSA($\lambda$) algorithm, and they are tuned in the same way.

6.7.1 NFQ-SARSA($\lambda$) for Mountain Car

As discussed in section 6.6.2, the mountain car problem does not use the most optimal input representation, I will, however, continue to use this non-optimal representation, so that the results can be directly compared to that of the $Q$-SARSA($\lambda$) algorithm. With the parameters tuned at $\alpha = 0.9$, $\epsilon = 0.01$, $\gamma = 1.0$, $\lambda = 0.4$, $\sigma = 0.3$, a cache size of 500 and a cache commit interval of 20, a policy that could reach the goal in an average of 125 steps in the 1000 off-line episodes was produced. This is significantly better than the 146 steps learned by the cascading $Q$-SARSA($\lambda$) algorithm, and I believe that the primary reason for this is the fact that results from different runs can be combined to produce a better policy.

The results for the NFQ-SARSA($\lambda$) algorithm was obtained with only 1000 episodes of learning as oppose to the 10,000 episodes for the $Q$-SARSA($\lambda$) algorithm. When mentioning the 1000 episodes compared to the 10,000 episodes, the time aspect must be kept in mind, since this was the primary reason for only allowing 1000 episodes for the cascading NFQ-SARSA($\lambda$) algorithm. The 10,000 episodes
of learning and 1000 off-line episodes took 7 minutes and 50 seconds for the cascading \(Q\)-SARSA(\(\lambda\)) algorithm, while the 1000 episodes of learning and 1000 off-line episodes only took 1 minute and 36 seconds for the cascading NFQ-SARSA(\(\lambda\)) algorithm. This clearly shows that when only 1000 episodes is used for learning, time is not a problem for the NFQ-SARSA(\(\lambda\)) algorithm. The relatively short time for the NFQ-SARSA(\(\lambda\)) algorithm, compared to the larger time for the \(Q\)-SARSA(\(\lambda\)) algorithm, should not be seen as an indication that the alleged time consummation of the NFQ-SARSA(\(\lambda\)) algorithm is false. It should rather be seen as an indication that the early episodes are faster than the later episodes, partly because full training does not start before the cache is full, and partly because the training of the neural network takes longer time as more candidate neurons are installed in the network. As a comparison, the cascading \(Q\)-SARSA(\(\lambda\)) algorithm only use 19 seconds for 1000 episodes of learning and 1000 off-line episodes.

Figure 6.28 shows the variations in performance for the \(\alpha\), \(\epsilon\), \(\gamma\), \(\lambda\) and \(\sigma\) parameters, and although the NFQ-SARSA(\(\lambda\)) and \(Q\)-SARSA(\(\lambda\)) algorithms are quite different in the way they train the network, the graphs for the two algorithms are very similar.

The main difference between the performance of the parameters for the NFQ-SARSA(\(\lambda\)) algorithm and the performance for the \(Q\)-SARSA(\(\lambda\)) algorithm, is the fact that the NFQ-SARSA(\(\lambda\)) algorithm is less dependent on the \(\epsilon\) parameter, while it is more dependent on the \(\gamma\) parameter. It is hard to say why there is a stronger dependency to the \(\gamma\) parameter, but I believe the reason that the number of explorative steps are not that important, is that NFQ-SARSA(\(\lambda\)) can combine results from different runs.

Figure 6.29 shows the performance as a parameter of the cache size and cache commit interval. Again here, the graphs are very similar to the graphs in figure 6.21 on page 122, which show the same performance for the cascading \(Q\)-SARSA(\(\lambda\)) algorithm. However, the cascading NFQ-SARSA(\(\lambda\)) algorithm is able to produce good policies for a broader spectrum of cache sizes and cache commit intervals.

Since the same cache size and cache commit interval are used, for the cascading NFQ-SARSA(\(\lambda\)) algorithm as for the cascading \(Q\)-SARSA(\(\lambda\)) algorithm, the network also grows at approximately the same rate. The consequence of this fact, is that fewer neurons are added to the network trained by NFQ-SARSA(\(\lambda\)), since fewer episodes are used for training. At the end of the 1000 episodes, only two candidate neurons are added to the network, one after 460 episodes and one after 970 episodes. With only 1000 learning episodes and a neural network with 2 hidden neurons, it is amazing that such good performance can be achieved by the NFQ-SARSA(\(\lambda\)) algorithm.

### 6.7.2 NFQ-SARSA(\(\lambda\)) for Cart Pole

For the mountain car problem, the cascading NFQ-SARSA(\(\lambda\)) algorithm has shown true potential. This section will explore if this potential can be expanded to the cart pole problem. The cascading \(Q\)-SARSA(\(\lambda\)) algorithm was very effective for this problem, and was able to produce a policy that could balance the pole for 300 steps after around 300 episodes. This is very difficult to beat, since the cache should be filled before the neural network can be trained with the cache, and since at least 500 epochs of training must be completed before any candidates can be installed. For the cascading \(Q\)-SARSA(\(\lambda\)) the candidate was installed very early after this point, so there is really no room for improvement in that area. In order for the cascading NFQ-SARSA(\(\lambda\)) algorithm to perform better than the cascading \(Q\)-SARSA(\(\lambda\)) algorithm, it should not produce a better policy in shorter time, but instead be more tolerant about how the parameters are set, and produce good policies for a broader range of parameters.
Figure 6.28: Average performance in the on-line and off-line case, for the NFQ-SARSA(λ) trained mountain car problem, for variations of the α, ε, γ, λ and σ parameters, with the remaining parameters fixed at their tuned values. The on-line and off-line performance are measured after 10,000 episodes, where the off-line performance is an average of 1000 episodes.
The cascading NFQ-SARSA(\(\lambda\)) was able to produce a policy that was able to balance the pole for 300 episodes in all of the 1000 off-line episodes, and which achieved an average off-line reward of 310, with the tuned parameters fixed at \(\alpha = 0.7, \epsilon = 0.1, \gamma = 1.0, \lambda = 0.9, \sigma = 1.0\), a cache size of 750 and a cache commit interval of 5. This is almost as good as the cascading Q-SARSA(\(\lambda\)) algorithm, but what really sets this policy apart from the policy learned by the Q-SARSA(\(\lambda\)) algorithm is the number of installed candidates. The cascading Q-SARSA(\(\lambda\)) algorithm only installed one candidate, but the cascading NFQ-SARSA(\(\lambda\)) algorithm has installed 23 candidates to produce a similar policy.

The reason that only one neuron was installed by the Q-SARSA(\(\lambda\)) algorithm, was that the candidate was installed to precisely match the neural network, and that the MSE dropped below 0.0001 after that. For the NFQ-SARSA(\(\lambda\)) algorithm the first installed candidate was not as successful, and the MSE did not drop below 0.0001. There are several different reasons for this, and I believe one of the reasons that the MSE did not drop as drastically, is the fact that the NFQ-SARSA(\(\lambda\)) algorithm recalculates the data that the neural network is trained with, for each epoch. The Q-SARSA(\(\lambda\)) algorithm does not do this, and only removes the oldest elements and inserts new, which with the parameters used for the cascading Q-SARSA(\(\lambda\)) algorithm meant that 1995 elements out of 2000 was the same between two epochs. With this in mind, it is very hard to see any way that the MSE could drop this low for the NFQ-SARSA(\(\lambda\)) algorithm. However, the fact that the MSE did not drop, does not explain why the first candidate was not able to produce a good policy. One explanation for this could be the cache size. As can be seen in figure 6.30, the cache size with the best off-line performance is 750. I do not believe that 750 is a large enough cache to train a candidate, which will be able to fit exactly into the neural network, and produce a network that can balance the pole for 300 steps, and a closer look at the individual episodes also reveals that the first time the agent was able to balance the pole for the full 300 steps, was after 480 episodes where 4 candidates were installed.

This explains why the tuned parameters did not produce a candidate neuron which was able to fit exactly into the empty neural network and produce a policy that could balance the pole for 300 steps, but it does not explain why the agent with the cache size of 2000 was not able to do so. The NFQ-SARSA(\(\lambda\)) algorithm should have just as large a chance of producing such a candidate as the Q-SARSA(\(\lambda\)) algorithm. When looking at the performance for different cache sizes in figure 6.30,
6.7. **NEURAL FITTED Q-SARSA(\(\lambda\))**

Figure 6.30: Average number of steps in the on-line and off-line case, for the NFQ-SARSA(\(\lambda\)) trained cart pole problem, for variations of the cache size and the cache commit interval, with the remaining parameters fixed at their tuned values. The on-line and off-line number of steps is measured after 10,000 episodes, where the off-line steps is an average of 1000 episodes.

and comparing them to the performance for the cascading Q-SARSA(\(\lambda\)) algorithm displayed in figure 6.25 on page 126, the on-line performance increases on both graphs as the cache size increases. This suggest, that perhaps the agent with the cache size of 2000 was able to produce a candidate neuron, which would fit into the network and allow the pole to be balanced for 300 steps, but that later installed neurons made the network forget the policy. An initial look at the installed candidate neurons over time, and the cumulative reward over time in figure 6.31 does not suggest that this is the case. There are some small jumps in the cumulative reward when some of the candidates are installed, but nothing that lasts for more than a couple of episodes.

However, a close look at the individual episodes, reveals that when the first candidate neuron is installed, the pole is balanced for 300 steps in two episodes, and when the second candidate is installed, the pole is balanced for 300 steps in three episodes. The same pattern is repeated when many of the candidates are installed, which means that the NFQ-SARSA(\(\lambda\)) algorithm is able to produce candidates that can be installed in the neural network so that the resulting policy can balance the pole for 300 steps. The question is then, what distinguishes NFQ-SARSA(\(\lambda\)) from Q-SARSA(\(\lambda\)), in such a degree that the one algorithm is able to use the installed candidate, while the other is not? I believe that the core to the answer of this question lies in the recalculation of the \(Q(s, a)\) values, made by the NFQ-SARSA(\(\lambda\)) algorithm. When no recalculation is done, the \(Q(s, a)\) values that was used to train the candidate will remain in the cache for some time after the candidates have been installed, only slowly being replaced with new \(Q(s, a)\) values, which means that the neural network will be given time to converge to these values. However, when the \(Q(s, a)\) values are being recalculated before each training epoch, the convergence will not happen so fast, and as the data slowly leaves the cache, the good policy will be forgotten. This means that although the NFQ-SARSA(\(\lambda\)) algorithm is able to produce candidates that can greatly improve the performance, it is not able to use them properly, and it will have to rely on slower converge with more candidates in order to produce a good policy.

With this in mind, the performance for variations of \(\alpha, \epsilon, \gamma, \lambda\) and \(\sigma\) parameters in figure 6.32, can be investigated a bit further in order to see how tolerant the NFQ-SARSA(\(\lambda\)) algorithm is to changes in these parameters.

The overall off-line performance for the parameters in figure 6.32 is a bit worse than the performance for the cascading Q-SARSA(\(\lambda\)) algorithm in figure 6.25 on
CHAPTER 6. REINFORCEMENT LEARNING TESTS

Figure 6.31: At the top, the number of installed candidates over time, for the NFQ-SARSA(λ) trained cart pole problem, and at the bottom the cumulative reward over time, for different cache sizes.
6.7. **NEURAL FITTED Q-SARSA($\lambda$)**

Figure 6.32: Average number of steps in the on-line and off-line case, for the NFQ-SARSA($\lambda$) trained cart pole problem, for variations of the $\alpha$, $\epsilon$, $\gamma$, $\lambda$ and $\sigma$ parameters, with the remaining parameters fixed at their tuned values. The on-line and off-line number of steps is measured after 10,000 episodes, where the off-line steps is an average of 1000 episodes.
This is largely controlled by the fact that several of the $Q$-SARSA($\lambda$) agents was able to install one candidate, and balance the pole in 300 steps from there on. The on-line performance is also affected by this, since the agents with the one candidate neuron would be able to balance the pole for 300 steps, in most of the thousands of episodes after the candidate have been installed. Even if the NFQ-SARSA($\lambda$) algorithm was able to do this, it would only have a couple of hundred episodes to balance the pole, and the on-line reward would never be able to be as high as for the $Q$-SARSA($\lambda$) algorithm.

Besides the overall performance differences, there are also striking differences between the performance for variations of the $\alpha$ and $\epsilon$ parameters. Where the $\alpha$ and $\epsilon$ parameters should be small to produce good results for the $Q$-SARSA($\lambda$) algorithm, they shall be large to produce good results for the NFQ-SARSA($\lambda$) algorithm. It is not surprising that there are differences in the parameters, since the $Q$-SARSA($\lambda$) algorithm was tuned to find one profitable candidate neuron and install it in the network, and the NFQ-SARSA($\lambda$) algorithm is tuned to slowly converge as more neurons are installed. I believe that the small $\epsilon$ and $\alpha$ parameters for the $Q$-SARSA($\lambda$) algorithm, are especially well suited for the converge phase immediately after the candidate has been installed, where it is important that the network does not forget too fast. Similarly, I believe that the large $\alpha$ parameter is vital to the NFQ-SARSA($\lambda$) algorithm, because it allows for the recalculated $Q[s,a]$ values to persist themselves in the neural network. Although the performance of the NFQ-SARSA($\lambda$) algorithm is not as good as for the $Q$-SARSA($\lambda$) algorithm it is still just as good as the results from the NIPS conference.

While the cascading NFQ-SARSA($\lambda$) is superior to the cascading $Q$-SARSA($\lambda$) algorithm for the mountain car problem, the opposite is the case for the cart pole problem, so it is not easy to determine which of the two that is the best algorithm. It is clear to see that the NFQ-SARSA($\lambda$) has some advantages, but it is slower than $Q$-SARSA($\lambda$) and it can be hard for the underlying neural network to convergence, since the values that it should be trained with are recalculated before each training epoch.

### 6.8 Backgammon

The tests in section 6.4 to section 6.7 have shown that the cascading $Q$-SARSA($\lambda$) and cascading NFQ-SARSA($\lambda$) algorithms can produce good results for the mountain car and cart pole problems. The tests also show that the two algorithms perform better than the incremental $Q$-SARSA and the batch NFQ-SARSA($\lambda$) algorithm. However, the mountain car and cart pole problems are relatively simple problems, which could also be seen in the compact networks, that was created by the algorithms. In order to test how well the algorithms scale, a larger more complex problem must be used for testing.

The backgammon problem is a large complex problem, where an agent must learn to play backgammon well. There are several different reasons as to why the backgammon problem is particular interesting, with the most important being that it was one of the first large scale problems, for which reinforcement learning was successfully applied (Tesauro, 1995). Another important reason, is that although it is relatively easy to model, it is very difficult to solve by using some of the brute force methods which have been successful for games such as chess. The reason that these methods are not very successful for backgammon, is that random dice rolls are used which means that the possible number of next states is very large, and this makes it very difficult to do brute force lookahead.
6.8. BACKGAMMON

6.8.1 Problem Description

The backgammon problem is briefly discussed in section 6.1.4, and the rules of the game are available at backgammon sites like e.g. www.bkgm.com. The opening board position is displayed in figure 6.33. Backgammon can be played as a single game, but most tournaments are played to a fixed number of points. When using points, a double cube is used, which provides the possibility to earn more points in a game. The use of the double cube is not very complicated, but it does provide an extra aspect which will make it harder for the agent to learn. For this reason I have chosen to leave the cube out of the problem, which is coherent with the work of Bellemare et al. (2004).

Figure 6.33: The opening position for backgammon, with indication of the home board and the outer board. Red must move clockwise and bear of the checkers after they have all been positioned in red’s home board, while white must move counter clockwise.

Since the game can use board positions as after-states the agent is presented with a number of possible after-states, and it will have to determine which of the given after-states that is most profitable. Many different representations for the board positions can be selected and appendix B include a discussion of different representations, but in order to compare my implementation with the implementation of Bellemare et al. (2004) their representation will be used, which according to Sutton and Barto (1998) is also the representation used by the first version of TD-Gammon. However, as described in appendix B this representation is far from ideal and I doubt that this was the actual representation used by TD-Gammon.

6.8.2 Agent Setup

For the problems discussed in section 6.2 the agent setup was pretty straightforward, and the agent was simply set to interact with the environment and learn from the experience. In the two player game of backgammon, the dynamics of the environment are controlled by the opponent, which makes the situation more complex.

The problem can be viewed as an ordinary reinforcement learning problem, if the opponent is a fixed player and the agent can be set to play against this opponent and learn from that experience. However, when the opponent is fixed, the agent will only learn how to beat this opponent and there is no guarantee that the policy learned will be useful in the general case. This is especially a problem if the fixed opponent is a weak opponent, since the strategies needed to beat this opponent will probably not be sufficient to beat an expert human player. However, if the opponent plays at an expert level, the learning will be very slow, since the agent will have a very hard time winning any games, and even in this case, the agent will only learn to beat this particular opponent. In order to speed up learning, the
opponent could be made variable, so that it will gradually be better and better. In this way the agent will win and loose games all the way through the learning phase, and the learning will be maximized. This strategy seems very promising, but it requires that there already exists an expert level player that can be used as opponent. This poses a problem: If the expert player does not exists, the strategy cannot be used, and if the expert player does exists, there might not be much point in trying to build a new expert player.

When an expert opponent does not exist, reinforcement learning can be used to train one. This can be done by having two agents which play against each other and gradually improve their play. When playing with two separate agents, the agents will develop strategies to beat each other and gradually become better at the game. However, since the agents learn separate policies there may be serious problems with convergence. The convergence problem lies in the fact that when the agent \( A \) is winning, agent \( B \) will try to find a policy that will beat agent \( A \), and when this has been achieved, agent \( A \) will try to find a policy that will beat the new \( B \) agent. This makes for an elaborate dance where the environment is constantly changing, and the agents try to incorporate the change into their policy. While incorporating the change, the agents will tend to over-optimize for the change and hence forget earlier learned strategies. This ever changing target is a variation of the moving target problem, and it will mean that convergence will be problematic. With that said, I do believe that given enough time, in most cases the policies for the two agents will start to converge to an efficient policy. However, in some cases I believe that one of the agents will begin to be much better than the other, and the other agent will end up having a very difficult time learning because it only seldom wins games that it can learn from. In this case, the final learned policy will be very suboptimal for both agents.

One way of resolving some of these problems, is by making the agents update and use the same policy. This way the policy will learn from a winning and a loosing game in each round. This has several advantages compared to using separate policies:

- There is no risk of one policy becoming significantly better than the other, since there is only one policy, and the training samples will always be uniformly distributed, so there will be no chance that the policy will be biased towards a positive or a negative result.

- The problem of the ever changing target will not be as significant, since the opponent is always exactly as good as the agent. This means that the policy will constantly be optimized to be a little bit better than the last learned policy. There is still a chance that the final policy will be stuck in a local minimum, but the moving target problem will not be as significant.

- Since the policy is learning from both the winning and the loosing game, the number of training samples will be twice that of the separate policies.

This method of learning through self-play is the same method as was used by Tesauro (1995) in the TD-Gammon game. Studies by Pollack et al. (1997) indicates that it was exactly this method of self-play that was the central reason that Tesauro had such a success with TD-Gammon, and that the TD(\( \lambda \)) algorithm in itself was not as important. Self-play was also used in all the experiments by Bellemare et al. (2004).

When learning through self-play, it is not possible to measure the on-line performance, since the agent will always win 50% of the games. I will instead measure the off-line performance against a random player and against a heuristic player, in the same way as done by Bellemare et al. (2004), which will allow my results to be
6.8. BACKGAMMON

directly compared with their results. The heuristic used is simple and is mainly based on the pip count (see appendix B), but still discourages blots and encourages building blockades.

For the mountain car and the cart pole problems, the \(Q(s,a)\) values that the neural network should learn would change over time, since the reinforcement learning algorithm would learn new and more accurate values. This poses some challenges for the neural network training algorithm, and these challenges are also present for the backgammon problem. However, the backgammon problem adds an additional challenge, since the environment is also changing, whenever the opponent changes. And since self play is used, the opponent changes for every step. This means that some of the experience achieved in the early learning, will not be valid in the later episodes, and it also means that some of the oldest steps in the cache may not be valid when they are used for training. This should not be a huge problem, as long as the agent only changes slowly, and I believe that the benefits of self-play far outweighs the disadvantages.

6.8.3 Backgammon Test Setup

Bellemare et al. (2004) tested an online incrementally trained agent and an agent trained using cascading neural networks. The online version greatly outperformed the cascading agent, and looking at the results of Bellemare et al. (2004) it is clear that the neural network grows far too quickly for any significant learning to take place, and I believe that this is the main reason for the bad performance. After 1000 games all configurations had more than 70 hidden neurons, and after 10,000 games all configurations had more than 260 hidden neurons.

The sliding window cache does not suffer from this problem to the same extent, and I believe that it will be able to perform better than the implementation by Bellemare et al. (2004), and I also believe that the implementation will perform better than the incrementally trained version.

The implementation by Bellemare et al. (2004) was able to produce an incrementally trained agent in 10,000 games that was able to win in approximately 99% of the games against a random player and in 73% of the games against the heuristic player described in section 6.8.2. The similar cascade trained agent was only able to win in approximately 95% of the games against a random player, and in 11% of the games against the heuristic player.

The mountain car and cart pole problems was tested using the incremental, batch and cascading \(Q\)-SARSA(\(\lambda\)) algorithm, as well as with the cascading NFQ-SARSA(\(\lambda\)) algorithm. This was done in order to compare the different algorithms, and the results showed that the cascading \(Q\)-SARSA(\(\lambda\)) and the cascading NFQ-SARSA(\(\lambda\)) were generally the best performing algorithms. For the backgammon problem, the main goal is to test how well, the cascading \(Q\)-SARSA(\(\lambda\)) and the cascading NFQ-SARSA(\(\lambda\)) algorithms scale, so it is not necessary to test all the different algorithms. However, there need to be some tests that can be used for comparison. Ideally these tests would include the incremental and batch \(Q\)-SARSA(\(\lambda\)) algorithms, but this is not really needed, since the results by Bellemare et al. (2004) is based on the same backgammon implementation and may be used as a baseline. I have chosen to use only the results by Bellemare et al. (2004), partly because I feel that they are sufficient for the purpose, and partly because the backgammon benchmarks are extremely time consuming. The backgammon problem is more time consuming than the other problems for three reasons: Firstly the number of inputs to the neural network is much larger meaning that the neural network takes more time. Secondly, the number of available actions in each step is larger than for the other problems and lastly, the backgammon environment is a more complex environment, which is executed as a separate Java application that communicates with
the agent through a pipe. This combined means that learning for 10,000 games of backgammon can easily take one to two hours (on a 3GHz P4 machine) or more if many candidate neurons are added, which in turn means that testing all the possible values for a single parameter takes more than a day and tuning the entire set of parameters can take several weeks. While I have chosen not to focus on the incremental and batch algorithms, I have chosen to focus more on the cascading algorithms, so as to explain, which circumstances that has an influence on the scaling ability of the algorithm.

6.8.4 Test Results for \(Q\)-SARSA(\(\lambda\))

As with the mountain car and cart pole problems, the parameters are also tuned for the backgammon problem, in order to produce the best performing policy. The tuned parameters are \(\alpha = 0.1\), \(\epsilon = 0.001\), \(\gamma = 1.0\), \(\lambda = 0.4\), \(\sigma = 0.4\), a cache size of 500 and a cache commit interval of 50. Using the tuned parameters an agent was trained for 10,000 episodes, and was able to learn a policy that could win in 99.75% of the 1000 off-line episodes against the random player, and in 71% of the games against the heuristic player. This is close to the results of the incremental agent by Bellemare et al. (2004), and it is much better than the cascading results by Bellemare et al. (2004), but not nearly as good as expected. However, a closer look at the final neural network which will be presented later in this section, reveals that the results are actually surprisingly good. Figure 6.34 shows the percentage of wins, for variations of the \(\alpha\), \(\epsilon\), \(\gamma\), \(\lambda\) and \(\sigma\) parameters.

An interesting property of the parameters in figure 6.34 is that the algorithm can learn to beat the random player in 99% of the games for many different parameters, which is also why it can be hard to see the random graph for the \(\epsilon\), \(\lambda\) and \(\sigma\) parameters. Another interesting property is, that the high stochasticity of the backgammon problem means that the \(\epsilon\) parameter can be kept small. It is also interesting how little influence the \(\lambda\) parameter has on the performance, which seems to indicate that eligibility traces are not of much use in this problem. It is hard to say why this is the case, since the delayed reward of the problem indicates that eligibility traces should be beneficial.

While the parameters in figure 6.34 are not too interesting, the cache size and cache commit interval in figure 6.35 are much more interesting. The cache is relatively small at 500, and the commit interval is large at 50. This means that every training sample is only used for training 10 times, and that the network is only trained approximately once per game.

Why the optimized cache size and cache commit interval is this small is difficult to determine, and it might be that the parameters have been tuned into a local minimum. I do, however, believe that the reason is closely related to the growth of the network. When the cache size is small and the cache commit interval is large, the network grows very slowly. Figure 6.36 shows exactly how slow the network grows; after 10,000 games only three candidates have been installed.

It is very surprising that a neural network with only three hidden neurons is able to produce an agent that performs as well as the agent actually does. Especially since the agent by Bellemare et al. (2004) used 40 hidden neurons, which is also the number of hidden neurons in the first version of TD-Gammon. However, as mentioned earlier I believe that the small number of installed candidate neurons is exactly what made the cache size and cache commit interval tune to these values. I believe that this is the case, because the installed candidates are only of little use to the final network. If the installed candidates are only of little use, the parameters will be tuned so that only a few candidates are installed, which explains why the parameters are tuned as they are. The claim that the installed candidates are only of little use, can be supported by trying to train the tuned parameters for more
Figure 6.34: Average off-line win percentage against the random and heuristic opponent, for the cascading Q-SARSA($\lambda$) trained backgammon problem, for variations of the $\alpha$, $\epsilon$, $\gamma$, $\lambda$ and $\sigma$ parameters, with the remaining parameters fixed at their tuned values. The win percentage is measured after 10,000 games, as an average of 1000 games.
than 10,000 games, and observe what happens as more neurons are added. Trying to train the agent for 50,000 games showed that the performance started to decline when more candidates are added after 10,000 games, and the final percentage of wins against the heuristic player after 50,000 games is close to 40%. This explains why the parameters are tuned not to add many candidates, but it does not explain why the candidates are not beneficial.

I do not believe that 500 samples are enough to train truly effective candidates for a complex problem such as backgammon, and although the performance was surprisingly good with only three candidate neurons, I believe more candidate neurons are needed in order to achieve better performance, and that a larger cache should be used to train the candidates. However, although the results from figure 6.36 shows that enlarging the cache will add more candidate neurons to the network, figure 6.35 clearly shows that this will also degrade the performance. It is hard to say why this is the case, but one reason could be that the opponent is constantly changing which also means that the $Q^*(s,a)$ values are constantly changing, which in turn means that the candidates installed early in the training will not be relevant later on in the training. This gives some explanation as to why the installed candidates are not beneficial and the next section will discuss what could be done in order to avoid this problem.

**Improving the Performance of Cascading $Q$-SARSA($\lambda$)**

Chapter 4 and 5 suggest several methods of improving the performance for reinforcement learning combined with cascading neural networks. Some of these methods will be discussed here, so as to explore if they might help improve the performance for the backgammon problem, and especially if they might help the backgammon problem achieve better performance when more candidate neurons are added to the neural network.

Section 5.6 suggests one way to get around the problem of ensuring that candidates are beneficial, even if the $Q^*(s,a)$ function has changed since the candidate was installed. This method is the whole ANN method of training which is shown in algorithm 7 on page 87. This method did not produce good results for the mountain car and cart pole problems, but it may prove to be more beneficial for the backgammon problem. Another technique which may improve performance is tabu search from section 4.6.1. This technique has not been tested on the mountain car and
Figure 6.36: The evolvement in the number of installed candidates, for the cascading \( Q\)-SARSA(\( \lambda \)) trained backgammon problem, for variations of the cache size and cache commit interval parameters, with the remaining parameters fixed at their tuned values.
cart pole problems, since it only makes sense when the same states are visited several times, which only happens very seldom for continuous problems. However, the backgammon problem is a discrete problem, and although there are many possible states, the opening and closing states are likely to be visited several times, which suggests that tabu search may help improve the performance for the backgammon problem.

When the cascading Q-SARSA(\(\lambda\)) algorithm was used to learn the cart pole problem, one of the deciding factors for the great performance was the large cache, but the performance was mostly contributed to the large cache that was used for training the candidates, and less to the large cache which was used to train the neural network. This led to the suggestion of allowing a different cache size to be used when the candidates are trained and when the neural network is trained. This method might prove useful for the backgammon problem, especially since figure 6.35 shows that simply enlarging the cache size will not improve the learning.

Section 6.8.5 will show how a combination of whole ANN training and a larger cache can lead to increased performance, while appendix C will take a closer look at tabu search, rehearsing and using a larger cache for training the candidates and explain why these modifications was not able to increase performance for the backgammon problem. Another discussion which was brought up by the tests, is the issue about the dependency between the cache size, cache commit interval and the growth of the network. This issue is discussed further in section 8.1, and will not be taken into consideration during the tests.

6.8.5 Test Results With a Larger Cache

In order to obtain better results for the backgammon problem, it is important to install more candidates and that a larger cache is used to train the candidates. For this reason cache sizes up to 10,000 is included instead of the normal 2000 and whole ANN training is used. For convenience I will refer to this tuning session with a larger cache as tuning session B and I will refer to the first tuning session for the backgammon problem as tuning session A.

In order to obtain the desired results, the cache size is initially kept large, while the cache commit interval is kept small, and only the remaining parameters are tuned. This is done, to prevent the tuning from ending in a local minimum, where the cache size and the number of installed candidates are small, since this was identified as one of the key factors prohibiting learning. After a couple of iterations, the cache size and cache commit interval are allowed to be tuned and the iterations are continued until the tuning converges. When the tuning converged, the cache size was 4000 and the cache commit interval was 20, which means that every step is used 200 times for training. This is 20 times more often than the 10 times, which was used for tuning session A and the extra training means that a policy was learned, that was able to win 78% of the 1000 off-line games against the heuristic player. However, this policy was learned after 8000 episodes and the best performing agent after 10,000 episodes was only able to win 74% of the games. A win percentage of 78% against the heuristic opponent is better than the 71% for tuning session A, and it is better than the 11% for the cascading agent by Bellemare et al. (2004) and it is 5 percentage points better than the incremental agent. Although 5 percentage points might not seem like much, it should be noted that even with a perfect policy it will not be possible to win 100% of the games against a heuristic player since the dices include an element of luck. With this in mind, an agent that can win 78% of the games against a heuristic player is a very capable player and 5 percentage points is a great improvement. Still, in order for a clear tendency to be visible, a larger improvement should be observed. While it is not possible to say that the cascading Q-SARSA(\(\lambda\)) algorithm performs significantly better than the incremental algorithm for larger
problems, it is possible to say that the cascading $Q$-SARSA($\lambda$) algorithm scales just as well or better than the incremental algorithm. This is the first time that such a claim can be made about an algorithm that combines cascading neural networks with reinforcement learning, and this alone is a big step for introducing more advanced neural network training algorithms in the field of reinforcement learning.

When comparing the cascading $Q$-SARSA($\lambda$) algorithm to the incremental results by Bellemare et al. (2004), there is a risk that the primary reason that the cascading $Q$-SARSA($\lambda$) is able to compete with the incremental algorithm, is that more tuning is used for the cascading $Q$-SARSA($\lambda$) algorithm. In order to test this, a tuning session was made with the incremental $Q$-SARSA algorithm. However, the tuned agent was only able to win in 68% of the games against the heuristic player, so the tuning does not seem to be a problem.

The test results for tuning session B produced better results, with parameters that are significantly different than the parameters used for the first session. The parameters used for the second session are $\alpha = 0.7$, $\epsilon = 0.01$, $\gamma = 1.0$, $\lambda = 0.3$, $\sigma = 1.0$, a cache size of 4000, a cache commit interval of 20 and in addition to this, whole ANN training is used. The most significant difference in parameters is the use of a larger cache and the use of whole ANN training, but also the low cache commit interval should also be noted. Figure 6.37 shows the average win percentage for variations of the cache size and cache commit interval.

![Figure 6.37: Average off-line win percentage, for the cascading $Q$-SARSA($\lambda$) trained backgammon problem with a larger cache, for variations of the cache size and cache commit interval parameters, with the remaining parameters fixed at their tuned values. The win percentage is measured after 10,000 games, as an average of 1000 games.](image-url)

The cache sizes from 500 to 2000 all show reasonably good results, but above this only a cache size of 4000 and 10,000 show good results, and for a cache size of 3000 and 8000 the results are actually much worse than a random player. Results this bad were not seen in the previous tuning session, and a closer look at the learning for these cache sizes are needed. A closer look reveals that the learning for both cache sizes are ended prematurely after 2000 to 3000 games, due to a maximum learning time of 5 hours, and that the reason for the long learning time, is the fact that the agents has played several games that lasts for several thousand steps, and in one occasion almost 70,000 steps, as oppose to the normal games of 20-60 steps. It seems like the agents have learned policies where they hit each other very often, but where they do not protect their checkers in any way. This leads to very long games and during these games several candidate neurons are added to the network as can be seen in figure 6.38, which leads to a serious degeneracy of the policy, that can not easily be corrected. The closer look also reveals that a cache size of 6000 also suffers from the same degeneracy, but not in the same degree.
This particular degeneracy was not expected, as it was thought that since the checkers move forward in each step, where they are not hit, the game will end after a limited number of steps. However, Tesaro (1995) document that a random agent may play games that last for several thousand steps, so although he have not experienced this particular degeneracy he does give hints that it may occur. A closer look at the agents that does not suffer from the degeneracy reveals that they also experience games that last a couple of hundred steps during the first 1000 games, where they have not learned a good policy yet. I suspect that the reason, that the agents with a large cache from the second tuning session is more likely to suffer from from this degeneracy, is the fact that they add candidates more quickly, so a long game will have a greater impact on the learned policy.

A closer look at the cache commit interval, reveals that except for a commit interval of 10, 30 and 40 which all suffer from the degeneracy, then the commit interval only has little influence on the actual performance. This is interesting, since that while the tuned cache commit interval of 20 adds 34 candidates, a cache commit interval of 80 only add 8 candidate neurons during the 10,000 games, but still manages to win 73% of the games against the heuristic player. This discovery suggests the same as the initial tuning, that many neurons are actually not needed in order to obtain good performance. I do, however, still suspect that some amount of hidden neurons is needed in order to obtain even better performance. This is supported by the results by Tesaro (1995) who used 40 and 80 hidden neurons for his different versions of TD-Gammon.

Adding whole ANN training to the training proved to consistently improve the off-line performance for several different cache sizes as can be seen in figure 6.39. This was not the case for tuning session A and although it is hard to give a precise reason, I believe that the increased number of candidate neurons and the increased $\alpha$ parameter is a key reason.

6.8.6 Test Results for NFQ-$\lambda$-SARSA

The cascading $Q$-SARSA($\lambda$) algorithm has shown an ability to scale to the large problem of backgammon, without many difficulties and it has shown that it can learn good policies while only using a very limited number of candidate neurons. The cascading NFQ-$\lambda$-SARSA($\lambda$) algorithm has shown very promising results, especially for the mountain car problem. This section will explore how the algorithm scales when used to learn the backgammon problem. The parameters for the NFQ-$\lambda$-SARSA($\lambda$) are tuned in the same way as for the other problems, which also means that a larger cache is not considered, and neither is rehearsing and tabu search. However, since the backgammon problem is a bit more complex, 4000 episodes of training is allowed instead of only 1000 episodes, and whole ANN training is also considered. The tuned parameters are $\alpha = 0.5$, $\epsilon = 0.2$, $\gamma = 0.7$, $\lambda = 0.5$, $\sigma = 1.0$, a cache size of 1000, a cache commit interval of 20 and no whole ANN training.

The cascading NFQ-$\lambda$-SARSA($\lambda$) algorithm was not able to show quite as good results as the cascading $Q$-SARSA($\lambda$) algorithm and the incremental algorithm by Bellemare et al. (2004). The tuned agent was actually only able to win in 17% of the games against the heuristic player, and 82% of the games against the random agent. However, a closer look revealed that one agent was able to win 23% of the games against the heuristic player after only 1000 games, so it did not seem like extending the learning period to 4000 games was beneficial. These results are far from the results of the cascading $Q$-SARSA($\lambda$) algorithm and the incremental algorithm, but before any conclusions are made about the scalability of the NFQ-$\lambda$-SARSA($\lambda$) algorithm, the individual parameters will be given a closer look. The win percentage for variations of the cache size and cache commit interval can be seen in figure 6.40.
Figure 6.38: The evolvement in the number of installed candidates, for the cascading Q-SARSA(\(\lambda\)) trained backgammon problem with a larger cache, for variations of the cache size and cache commit interval parameters, with the remaining parameters fixed at their tuned values. Some of the graphs are ended before the 10,000 episodes, because the training is ended after a fixed period of 5 hours.
Figure 6.39: Average off-line win percentage, for the cascading Q-SARSA($\lambda$) trained backgammon problem with a larger cache, for variations of the cache size parameter. The left graph shows variation where whole ANN training is applied, and the right show variations where whole ANN training is not applied. The win percentage is measured after 10,000 games, as an average of 1000 games.

Figure 6.40: Average off-line win percentage, for the cascading NFQ-SARSA($\lambda$) trained backgammon problem, for variations of the cache size and cache commit interval parameters, with the remaining parameters fixed at their tuned values. The win percentage is measured after 4000 games, as an average of 1000 games.
6.8. BACKGAMMON

For the cache size, there is not much difference in the performance for the small and the large cache. This is a bit surprising, since figure 6.41 shows that the smallest cache size of 100 only adds 4 candidate neurons, while the largest cache size of 2000 adds 36 candidates. While this is surprising, the obvious explanation must be that not much learning takes place, and that the structure of the underlying neural network has very little impact in the actual learning. This claim is supported by the fact that the cascading Q-SARSA(\(\lambda\)) algorithm was able to learn better policies with less candidates, and that the 23% wins which was achieved after 1000 games only used four candidates.

![Graph 1](image1.png)

![Graph 2](image2.png)

Figure 6.41: The evolution in the number of installed candidates, for the cascading NFQ-SARSA(\(\lambda\)) trained backgammon problem, for variations of the cache size and cache commit interval parameters, with the remaining parameters fixed at their tuned values. Some of the graphs are ended before the 10,000 episodes, because the training is ended after a fixed period of 3 hours.
Like the cache size, the cache commit interval has very little influence on the actual performance. However, for a cache commit interval of 10 or below, many candidates are generated very quickly and the performance suffers a bit from this.

The graphs for the \( \alpha \), \( \epsilon \), \( \gamma \), \( \lambda \) and \( \sigma \) are displayed in figure 6.42, and the only parameter which has a clear tendency is the \( \gamma \) parameter. A higher \( \gamma \) parameter means better performance, which is also visible in the fact that the tuned \( \gamma \) value is 0.7. The \( \alpha \) and \( \epsilon \) parameters which are usually very important for the performance, have very little impact on the final performance.

For the cascading Q-SARSA(\( \lambda \)) the whole ANN training proved to improve the training, so there was a hope that the same would be the case for the cascading NFQ-SARSA(\( \lambda \)) algorithm. However, nothing showed that this was the case, and the whole ANN training was only able to win 8% of the games against the heuristic player, and 79% against the random player.
There is a clear picture throughout the test of the cascading NFQ-SARSA(\(\lambda\)) algorithm on the backgammon problem. The picture is, that no matter how you change the parameters, not much learning occurs. Bellemare et al. (2004) was in the same situation, they tried several different modifications, but they were not able to achieve a reasonable performance. Their performance was even worse than the performance for the cascading NFQ-SARSA(\(\lambda\)) algorithm, and I believe that the reason that their performance was so bad, was that their combinational algorithm was simply not geared towards larger problem sizes. I believe that the cascading NFQ-SARSA(\(\lambda\)) algorithm has the same problem. The recalculation of the \(Q[s,a]\) values, in combination with the way that the cache and cache commit interval is handled does not work together in a way that can scale to larger problem sizes. Section 8.2 will discuss why the NFQ-SARSA(\(\lambda\)) algorithm was not able to scale and what can be done to scale the algorithm to larger problems and appendix D will give an in depth discussion of how beneficial the enhancements are, that the NFQ-SARSA(\(\lambda\)) algorithm has added to the NFQ algorithm.

6.9 Test Results

Many different aspects of the combination of reinforcement learning and advanced neural network training algorithms has been tested during the reinforcement learning tests. However, four key areas have been the primary focus, and these areas will also be the primary focus of the conclusion. The key areas are:

- \(Q\)-SARSA(\(\lambda\)) compared to \(Q(\lambda)\) and SARSA(\(\lambda\)).
- Incremental algorithms compared to batch and cascading \(Q\)-SARSA(\(\lambda\)).
- Cascading \(Q\)-SARSA(\(\lambda\)) compared to cascading NFQ-SARSA(\(\lambda\)).
- The scalability of the \(Q\)-SARSA(\(\lambda\)) and NFQ-SARSA(\(\lambda\)) algorithms.

It is not possible to compare the individual algorithms, without considering the aspect of scalability, so scalability will be handled as part of the individual algorithm comparisons.

6.9.1 Comparing \(Q\)-SARSA(\(\lambda\)) to \(Q(\lambda)\) and SARSA(\(\lambda\))

In order to do a thorough comparison between the \(Q\)-SARSA(\(\lambda\)) algorithm and the \(Q(\lambda)\) and SARSA(\(\lambda\)) algorithms, a closer look at the eligibility trace method is needed, since the eligibility trace method used for the \(Q\)-SARSA(\(\lambda\)) algorithm is not the same as is usually used for \(Q(\lambda)\) learning. However, for this thesis, the aspect of difference in eligibility trace methods has been left out, and it is assumed that the same eligibility trace method is used for all three algorithms. When this assumption is made, the comparison of the three algorithms can be made by simply looking at the \(\sigma\) parameter, which converts the \(Q\)-SARSA(\(\lambda\)) algorithm to \(Q(\lambda)\) when it is 0.0 and to SARSA(\(\lambda\)) when it is 1.0.

The \(\sigma\) parameter have been tested in a number of different situations, with the most thorough test made for the blackjack problem. For the blackjack problem there are not much difference between the performance of different values for the \(\sigma\) parameter, and this was a widespread phenomenon appearing in many different configurations of problems and algorithms. This was expected, since \(Q(\lambda)\) and SARSA(\(\lambda\)) often perform equally well for many problems.

While there were very little difference in the performance for many configurations, some showed huge differences. Several combinations of algorithms and problems showed interesting results, but the mountain car problem was especially
interesting, since it showed huge differences in performance for different $\sigma$ values for the four reinforcement learning algorithms. Figure 6.43 show the variations for the $\sigma$ parameter for the mountain car problem for all the algorithms; incremental $Q$-SARSA, batch $Q$-SARSA($\lambda$), cascade $Q$-SARSA($\lambda$) and cascade NFQ-SARSA($\lambda$).

![On-line and off-line performance for variations of the $\sigma$ parameter for the mountain car problem, for all the four different algorithms.](image)

For the incremental algorithm, there is a very clear tendency that only small values for $\sigma$ can produce a good result, meaning that the $Q$-learning is the best choice in this situation. Normally the structure of the problem would be examined to explain why $Q$-learning is more efficient than SARSA, but in this case looking at the structure of the problem will not help, since the same pattern is not repeated for the remaining three algorithms. The remaining three algorithms favor a larger $\sigma$ value, and there seem to be a tendency that a value around 0.3 to 0.4 is favored, although the batch and cascading $Q$-SARSA($\lambda$) also favor a value of 1.0, which is the SARSA($\lambda$) algorithm. It is very hard to explain why the different algorithms react this differently to the $\sigma$ value. The $\sigma$ value is closely linked to the $\epsilon$ value, since the value of $\sigma$ is only important when an explorative step is taken. However, the $\epsilon$ value can not be used to explain the lack of pattern. An $\epsilon$ value of 0.2 is used for the incremental and batch problems, and an $\epsilon$ value of 0.01 is used for the two cascading algorithms, which could indicate that the graphs should also be grouped in this way, but the two graphs that have the most in common is the batch and cascading $Q$-SARSA($\lambda$), so this is not the case.

The mountain car problem has shown that it is very difficult to predict which $\sigma$ value that will be the best in a given situation. Furthermore it has shown that although $Q(\lambda)$ and SARSA($\lambda$) are often equally good at solving a problem, there are situations where the one is much better than the other, and there are also situations where the $Q$-SARSA($\lambda$) algorithm is better than both the $Q(\lambda)$ and
6.9. TEST RESULTS

SARSA(λ) algorithms. The Q-SARSA(λ) algorithm is not only able to combine the Q(λ) and SARSA(λ) algorithms in a way that makes it easy to switch from the one to the other. It has also provided results that could not have been achieved with the two algorithms separated.

An important aspect of the Q-SARSA algorithm, which is very visual when looking at the four graphs in figure 6.43, is the fact that researchers can use the σ parameter to study trends, which have not been available when the Q(λ) and SARSA(λ) algorithms were separated. If you e.g. only look at the results for the two separate algorithms, for the batch algorithm, you would be led to believe that the SARSA(λ) is far better than the Q(λ) algorithm. By looking at the graph for the σ value you can see that this is not a clear trend, but for the incremental algorithm on the other hand there is a very clear trend. Without the full picture, you would not be able to see this, and you would not be able to make decisions based on this information.

6.9.2 Comparing Incremental to Batch and Cascading Q-SARSA(λ)

For both the cart pole and the mountain car problems, the batch algorithm was better than the incremental algorithm and the cascade algorithm was better than the batch algorithm. Likewise the cascading algorithm was better than the incremental algorithm for the backgammon problem. The tendency was not clear in all situations, and there is still an open issue of how well an incremental Q-SARSA(λ) algorithm with eligibility traces would perform. I do not believe that this algorithm can compete with the cascading Q-SARSA(λ) algorithm and the comparison with the incrementally trained agent by Bellemare et al. (2004) on the backgammon problem seems to support this claim. For the cart pole problem, the cascading algorithm showed that not only was it able to produce a much better policy than the incremental and batch algorithms, it was also able to produce a much more compact network.

The performance by the cascading Q-SARSA(λ) algorithm for the cart pole and the revisited mountain car problems, were actually so good that they could compare to some of the advanced approaches from the NIPS 2005 conference (Dutech et al., 2005). This is in itself interesting, but what really makes the algorithm stand out is that cascading Q-SARSA(λ) is also able to scale to the large problem of backgammon, which is something that is very difficult for the algorithm from the NIPS conference. This combination of very good performance for small problems, and scalability is a very attractive feature. The scalability is special, because it is the first time that scalability has been shown for a reinforcement learning algorithm that uses advanced neural network training algorithms, and because many of the advanced model-based and model-free algorithms have not been able to show the same scalability. On top of this the algorithm is often able to create very compact networks, that perform just as well as the larger networks used by the other algorithms.

It seems like the cascading Q-SARSA(λ) algorithm has it all, performance, scalability and small networks. This is also true, but the algorithm also has some issues that might make it difficult to apply the algorithm to a problem. The most important issue with this algorithm is the growth of the network in combination with the cache size and cache commit interval. The very strong connection between the cache size, the cache commit interval and the growth of the network means that the performance of the algorithm is also very dependent of these two parameters. Although it did not seem to be an issue for the tests in this thesis, this connection may prohibit learning in some situations, and it may also prohibit scalability. This problem will be discussed in section 8.1.
6.9.3 Comparing Cascading $Q$-SARSA($\lambda$) to NFQ-SARSA($\lambda$)

For the mountain car and cart pole problems, the two algorithms $Q$-SARSA($\lambda$) and NFQ-SARSA($\lambda$) had comparable results, and the $Q$-SARSA($\lambda$) was best for the cart pole problem, while the NFQ-SARSA($\lambda$) algorithm was best for the mountain car problem. However, when the algorithms tried to learn the backgammon problem, only the $Q$-SARSA($\lambda$) algorithm was able to show any considerable results. This is a setback for the NFQ-SARSA($\lambda$) algorithm, but as will be discussed in section 8.2, I believe that it is possible to scale the NFQ-SARSA($\lambda$) algorithm to larger problems, by applying minor modifications.

The tests in this chapter have only scratched the surface of these two algorithms, and I believe that further exploration of the two algorithms can help explain when the one algorithm will be most beneficial, and when the other will be. In order to make this exploration complete, the batch NFQ-SARSA($\lambda$) algorithm which have been left out of this test, needs also be tested, and both the batch NFQ-SARSA($\lambda$) and the batch $Q$-SARSA($\lambda$) algorithms will need to be tested for the backgammon problem.

The tests for the cascading $Q$-SARSA($\lambda$) algorithm have clearly showed, that it is possible to combine advanced neural network training algorithms with reinforcement learning, and create an algorithm that can perform just as well as model-based methods, while still being able to scale to larger problems. However, the tests also showed that in order to obtain good performance, some amount of tuning is needed, and no single set of parameters work well for all problems and algorithms.
Chapter 7

Conclusion

In this thesis I have analyzed, implemented and tested a reinforcement learning method which combines the novel model-free reinforcement learning algorithm of $Q$-SARSA($\lambda$), with the batch neural network training algorithm Cascade 2. This combination has provided very good results, and is able to scale to large problem sizes.

The combinational reinforcement learning algorithm is based on three elements; the neural network training algorithm Cascade 2, the model-free reinforcement learning algorithm $Q$-SARSA($\lambda$) and the sliding window cache which is a novel method for combining the two algorithms.

7.1 The Cascade 2 Algorithm

The Cascade 2 algorithm is a constructive neural network training algorithm, based on the Cascade-Correlation algorithm, that has only been described vaguely in literature, and it has not been thoroughly tested against incremental and batch neural network training algorithm. This thesis provides a thorough description and test of the original Cascade 2 algorithm and a slightly modified algorithm which uses iRPROP$^-$ as the internal weight update algorithm instead of the original Quickprop algorithm. The tests clearly show that especially the modified algorithm has excellent performance and it outperforms both incremental and batch algorithms. When comparing the Cascade 2 algorithm against the Cascade-Correlation algorithm, the Cascade 2 algorithm was able to solve the 8-Input parity problem in 38 epochs while only installing one candidate neuron as oppose to the 357 epochs and 4-5 candidate neurons for the Cascade-Correlation algorithm. I have implemented the Cascade 2 algorithm in the open source neural network library FANN, and by including the algorithm in this library it has been made available to the approximately 3000 people, who download the library every month. By making the implementation available it is my hope that researchers will use the implementation in their work, so that they can focus on complexities at a higher level.

7.2 The $Q$-SARSA($\lambda$) Algorithm

The novel $Q$-SARSA($\lambda$) algorithm is a natural combination of the standard model-free $Q(\lambda)$ and SARSA($\lambda$) reinforcement learning algorithms, that can be parameterized to resemble $Q(\lambda)$, SARSA($\lambda$) or a combination of the two. The algorithm was tested in many different situations and for most of these situations $Q$-SARSA($\lambda$) outperformed both $Q(\lambda)$ and SARSA($\lambda$). In the remaining situations $Q$-SARSA($\lambda$) was parameterized to resemble either $Q(\lambda)$ or SARSA($\lambda$) which means that the
performance of Q-SARSA(\(\lambda\)) is always just as good as or better than the separate \(Q(\lambda)\) and SARSA(\(\lambda\)) algorithms. For this reason I see the Q-SARSA(\(\lambda\)) algorithm as a natural successor to the \(Q(\lambda)\) and SARSA(\(\lambda\)) algorithms, and I see no reason to use the two algorithms separately.

\section*{7.3 The Sliding Window Cache}

Scaling reinforcement learning algorithms such as the Q-SARSA(\(\lambda\)) algorithm to larger problems, requires that a function approximator like a neural network is used. Traditionally only simple incremental neural network training algorithms have been used, because it is challenging to combine incremental reinforcement learning with batch neural network training such as Cascade 2. For this reason combinations of reinforcement learning and batch neural network training have only been implemented a few times before and the results have been mixed. It is especially problematic that it has not been possible to scale the combination to larger problem sizes, since scaling is the primary reason for using neural networks in combination with reinforcement learning.

For this thesis the novel sliding window cache has been developed that combines the Q-SARSA(\(\lambda\)) algorithm with either batch iRPROP\(^{-}\) or Cascade 2. The sliding window cache has been tested on the medium sized mountain car and cart pole problems, where it has been compared to results from the “Reinforcement Learning Benchmarks and Bake-offs II” workshop at the NIPS 2005 conference and to results for the Q-SARSA algorithm with incremental neural network training. The sliding window cache has also been tested on the large scale backgammon problem, where it has been compared to results from the literature. When comparing results of Q-SARSA(\(\lambda\)) to results from the NIPS conference it can not be expected that the Q-SARSA(\(\lambda\)) algorithm can compete, since most of the algorithms at the NIPS conference are highly optimized algorithms specifically targeted at small to medium sized problems such as the mountain car and cart pole problems.

The results of the mountain car and cart pole problems are summarized in table 7.1 and the results for the mountain car problem show that the Cascade 2 algorithm in combination with Q-SARSA(\(\lambda\)) is able to reach the goal in an average of 146 steps, which is better than the 155 steps for iRPROP\(^{-}\) and 165 steps for incremental training. As expected the cascading Q-SARSA(\(\lambda\)) algorithm was not able to compete with the 75-120 steps from the NIPS conference. However, a second run of the cascading Q-SARSA(\(\lambda\)) algorithm was executed using a different input encoding for the neural network and this run was able to reach the goal in an average of 110 steps, showing that the cascading Q-SARSA(\(\lambda\)) algorithm is able to compete with the results from the NIPS conference.

<table>
<thead>
<tr>
<th></th>
<th>Mountain Car</th>
<th>Cart Pole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental Q-SARSA</td>
<td>165</td>
<td>262</td>
</tr>
<tr>
<td>iRPROP(\lambda)</td>
<td>155</td>
<td>297</td>
</tr>
<tr>
<td>Cascade 2 Q-SARSA((\lambda))</td>
<td>146 (110)</td>
<td>300</td>
</tr>
<tr>
<td>NIPS 2005 Conference</td>
<td>75 - 120</td>
<td>0 - 300</td>
</tr>
</tbody>
</table>

Table 7.1: Average number of steps during 1000 off-line episodes, for the mountain car and cart pole problems. For the mountain car problem it is desired to have as few steps as possible. The result in parentheses was achieved using a different input encoding. For the cart pole problem it is desired to have as many steps as possible, where 300 is the maximum.

For the cart pole problem the cascading Q-SARSA(\(\lambda\)) algorithm learned a policy that was able to balance the pole in all of the 300 steps during 1000 test episodes,
which is better than the 297 steps for iRPROP$^-$ and the 262 steps for incremental training. The cascading $Q$-SARSA($\lambda$) algorithm was just as good as the best algorithms from the NIPS conference, but what really made the algorithm surpass all expectations, was that the exceptionally good result was achieved in only 300 episodes and that only one candidate neuron was installed by the Cascade 2 algorithm, making for an extremely compact neural network.

For the backgammon problem the sliding window cache was combined with the Cascade 2 algorithm and the agent was able to learn a policy that could win more than 99% of the games against a random player and 78% of the games against a heuristic player. This should be compared to earlier results from the literature where reinforcement learning combined with Cascade-Correlation was able to win 11% of the games against the heuristic player and a combination with incremental training was able to win 73% of the games. The results for the backgammon problem show that the sliding window cache can scale to large problems and that it can scale just as well or better than the incremental algorithm.

When the Cascade 2 algorithm was combined with reinforcement learning it was challenged, since it should build a network with an ever changing set of training data. The algorithm handled this challenge very impressively, and the results for the three problems clearly shows, that not only is the sliding window cache able to provide better results than using an incremental neural network training algorithm, but it is also able to scale to the large problem of backgammon. No earlier combinations of reinforcement learning and batch neural network training algorithms have been able to scale to larger problem sizes. The combination of results for smaller problems that can compete with the results from the NIPS conference and results for larger problems that can compete with incrementally trained neural networks is unique, because the algorithms from the NIPS conference can not scale to larger problems and the incrementally trained networks can not compete with the results from the NIPS conference.

7.4 The NFQ-SARSA($\lambda$) Algorithm

The Neural Fitted $Q$ Iteration (NFQ) algorithm has shown very promising results, but it requires data for all prior steps when the neural network is trained, which does not scale well. For this thesis I have enhanced the algorithm with several of the features from the $Q$-SARSA($\lambda$) algorithm. I have added eligibility traces and the combination of $Q$($\lambda$) and SARSA($\lambda$), changing the NFQ algorithm into the NFQ-SARSA($\lambda$) algorithm. I have also combined the algorithm with the Cascade 2 neural network training algorithm and I have made the algorithm more on-line so that it does not need data from all prior steps. The enhancements have increased the performance of the algorithm and the algorithm also functions well together with the Cascade 2 algorithm. For the mountain car problem the NFQ-SARSA($\lambda$) combined with Cascade 2 was able to reach the goal in 125 steps which was better than the cascading $Q$-SARSA($\lambda$) algorithm and for the cart pole problem it was able balance the pole for the complete 300 steps just like the $Q$-SARSA($\lambda$) algorithm. For these problems the NFQ-SARSA($\lambda$) algorithm performs as well or better than the $Q$-SARSA($\lambda$) algorithm. However, the NFQ-SARSA($\lambda$) algorithm was not able to scale towards the backgammon problem, where it was only able to win 23% of the games against the heuristic payer.

The NFQ-SARSA($\lambda$) algorithm has shown promising results, but convergence problems have meant that it could not scale to larger problems. In the future works chapter I will give some suggestions to how the NFQ-SARSA($\lambda$) algorithm may be scaled to larger problems and how the $Q$-SARSA($\lambda$) and NFQ-SARSA($\lambda$) algorithms may be merged to create an even better performing algorithm.
CHAPTER 7. CONCLUSION

7.5 Summary

The sliding window cache is the first method for combining reinforcement learning and advanced neural network training algorithms that has been able to scale to larger problems and it is an important progress within the field of reinforcement learning combined with neural networks. This thesis has thoroughly tested and analyzed the strengths and weaknesses of the sliding window cache, and the analysis reveals that the combination between reinforcement learning and advanced neural network training algorithms has the potential of achieving even greater performance and scalability. Some of the methods that might be used to exploit this potential is discussed in the future works chapter.
Chapter 8

Future Work

This thesis has provided several successful examples of combining model-free reinforcement learning with the Cascade 2 algorithm, and the results have shown that the combination is able to scale towards larger problem sizes. However, the work in this thesis is only one step on the road to achieving truly high performing and scalable reinforcement learning. This chapter will discuss some of the ways that the work in this thesis can be enhanced, so as to achieve better performance and scalability. This discussion will focus on three key areas, where there is room for improvement: The sliding window cache, improving the NFQ-SARSA(λ) algorithm and a method that enables the NFQ-SARSA(λ) algorithm to learn by example.

8.1 Improving the Sliding Window Cache

One of the largest problems with the cascading Q-SARSA(λ) algorithm is the fact that the cache size and cache commit interval has a huge impact on the growth of the network. Since the growth of the network is vital to the final performance of the agent, this means that these parameters are more used to control the growth of the network, than the actual performance of different cache sizes and cache commit intervals. This section discuss these problems and suggests methods of freeing the connection between the growth of the network and the cache parameters. A future project could experiment with these methods, and I am confident that such experiments will be able to enhance the performance of the cascading Q-SARSA(λ) algorithm.

The performance of the cascading Q-SARSA(λ) algorithm seem to be very dependent on the rate that candidate neurons are added to the network. If the neurons are added too quickly, the network will add many neurons that does not benefit the learning. When too many of these neurons are added to the network, it will be very difficult for the network to learn anything useful. If the candidate neurons are added too slowly, the network will not grow enough to be able to approximate advanced functions. The addition of new neurons are controlled by a series of parameters, and there also exists a series of parameters which influence the growth of the network, without directly controlling it.

The parameters which directly control the growth of the network are all parameters which have not been tuned during the tests in this thesis. These parameters are neural network parameters, and no neural network parameters have been tuned during the reinforcement learning tests. The reason that these parameters have not been tuned, are partly practical and partly conceptual. The practical part concerns the fact that there are already many parameters which must be tuned. Conceptually all of the parameters that have been tuned are reinforcement learning
parameters, and the neural network has been seen as a black-box in these tests, so for this reason the neural network parameters have not been tuned.

The neural network parameters which concerns the growth of the network are the patience parameter, the change fraction, the desired error and the min/max epochs. The patience parameter is the standard patience parameter, which is described in section 2.5.3 on page 23, and the patience parameter is dependent on the change fraction. The patience parameter determines the number of episodes that must pass with less than “change fraction” change in MSE, before it is decided that the learning has stagnated and a new candidate neuron should be added. The min/max epochs set a minimum and maximum on the number of epochs that may pass between adding new candidate neurons, and the desired error is a parameter which makes sure that no matter how many episodes have passed, a neuron will never be added when the error is below this limit.

In all of the benchmarks the patience parameter have been fixe at 15, the change fraction at 0.05, the desired error at 0.0001 and the min/max epochs at 500 and 5000 epochs respectively. With these parameters fixed, it can be believed that the network will grow with approximately the same speed in each test. When the cache size and cache commit interval are fixed, the networks will generally grow at approximately the same rate, but when these parameters are changed the network growth will change dramatically.

A larger cache size will generally mean that the change in MSE from one step to another will be smaller, simply because a larger percentage of the cache will be the same as in the previous step. When the change in MSE is smaller, the patience parameter will be more likely to decide that the learning has stagnated, and more candidate neurons will be added to the network. The same reasoning can be made about the cache commit interval. If the cache is trained after every step, only one data pattern will be changed between each training epoch, and the learning is more likely to stagnate. Furthermore, a smaller cache commit interval will mean that the network is trained more often and since there is a maximum on the number of epochs between neurons are added, neurons will be added at a higher rate. Figure 6.22 on page 123 shows how different values for these two parameters influence the network growth for the mountain car problem.

When these two parameters are so central for the growth of the network, this will actually mean that they will be optimized to control the growth of the network, and not optimized to control the learning. When it is known that these two parameters are very important for the performance of the batch algorithm, it is problematic that for the cascading algorithm, they are restricted by the fact that they influence the neural network growth.

The patience parameter is central part of the original Cascade-Correlation algorithm, and is designed to add a new neuron to the neural network whenever the training has stagnated. This makes good sense when the training data-set is fixed, but when the data-set changes from epoch to epoch, it could easily be more controlled by the change in the data-set, than by actual convergence. As a future project it would be interesting to investigate other methods for controlling the growth of the network. One method which could free the cache size and cache commit interval a bit, without changing the underlying Cascade 2 algorithm, would be to make the change fraction dependent on the cache size and cache commit interval and making the min/max epochs be dependent on the cache commit interval. By doing this, the cache size and cache commit interval can be changed, without changing the rate that the candidates are added to the network. It could also be interesting to investigate simpler methods, which could e.g. simply add a new candidate every n episodes. A method like this might not be as sophisticated, but it makes it very easy to control the growth of the network, and it removes all restrictions from the cache size and cache commit interval.
8.2 SCALING THE NFQ-SARSA(\(\lambda\)) ALGORITHM

More untraditional methods of controlling the growth of the network may also be considered. The current method is connected to the MSE of the network, which makes good sense in a neural network context, but since the network is used in a reinforcement learning context the growth could just as well be controlled by the learning, where new neurons are added when the average reward has stagnated instead of the average MSE.

Another problem which influences the learning and the growth of the network is the fact that for some problems very long episodes may occur. During a long episode more learning will occur than during a short episode, and several candidates may be installed during this episode. The mountain car and cart pole problems are cut off after 300 steps, so they do not suffer as much from this problem. This solution could also be applied to the backgammon problem, but I will suggest applying more general methods of handling these long episodes. One method of handling the episodes is to only allow learning to occur between episodes, meaning that the same amount of learning will be used for short episodes and for long episodes. One central problem for this approach is that for some reinforcement learning problems the agent may learn a policy which will send it into an infinite loop, meaning that the episode will never end, unless some new learning occurs. I suggest another possible method which does not suffer from this problem: For each time the neural network is trained within an episode, the cache commit interval should be doubled. By doing so, the very long episodes will only achieve a little amount of extra training. Another consequence of this approach is that more learning will happen in the beginning of the episodes. In the beginning of an episode, the cache will typically contain data from the end of the previous episode, and for many problems it is generally desirable to learn much about the end of an episode, because this will guide the agent to reach goals and avoid pitfalls.

Even if the cache size and cache commit interval is freed from influence on the growth of the network, these parameters still have a large influence on the performance, so a future project could look at other methods to implement these parameters. For the cache size, an interesting approach could be to not use an actual cache size, but instead use an annealing parameter. The annealing parameter would determine how much influence the individual data samples in the cache would have on the training of the neural network, where the most recent data samples would have the most influence. By applying a method like this, the actual cache size would not have to be determined, but samples would simply be removed from the cache when their influence was less than some threshold. This annealing parameter will work in much the same way as the \(\lambda\) parameter for the eligibility trace method.

8.2 Scaling the NFQ-SARSA(\(\lambda\)) Algorithm

The original NFQ algorithm uses the experience from all prior steps when training the neural network, and it recalculates all data in the cache before it trains the neural network. This is a very time consuming task, and the largest problem which has had successful results with this method is the keepaway soccer problem (Kalyanakrishnan and Stone, 2007). This problem has episodes of 10-20 steps, and it was only possible to train for 400 episodes in a day. The training was accomplished by letting the agent explore for 50 episodes, and use the experience from these 50 episodes to train the neural network for 1000 epochs. Then letting the agent explore for another 50 episodes and use the experience from the 100 episodes to train the neural network again, and continue this process for 400 episodes. If this number of steps is converted to the backgammon problem, this would mean that only 100-200 episodes could be used for learning, and since the agent plays both sides of the table, this will actually mean that only 50-100 games can be used.
I do not believe that 50-100 games is enough to learn a complicated problem such as backgammon, so a more on-line method is needed. The NFQ-SARSA(\(\lambda\)) algorithm implemented in this thesis used an on-line method which resembled the sliding window cache in many ways. However, mainly due to convergence problems, this method was not able to scale to the game of backgammon, which implies that other on-line methods are needed.

On-line methods can be implemented in many different ways, and I will generally split the methods into two groups: the methods that take the basic NFQ-SARSA(\(\lambda\)) algorithm and make it a bit more on-line, and the methods which combine the NFQ-SARSA(\(\lambda\)) with the Q-SARSA(\(\lambda\)) algorithm.

### 8.2.1 On-line NFQ-SARSA(\(\lambda\))

The method used to train the neural network by Kalyanakrishnan and Stone (2007) is actually not that different from the method used in this section, only the parameters differ dramatically. Since there are 10-20 steps in each episode for the keepaway soccer problem, this will mean that there are roughly 750 steps in 50 episodes, and since 1000 epochs of learning is used at this point, the NFQ-SARSA(\(\lambda\)) algorithm could be parameterized with a cache commit interval of 750 and then simply be trained for 1000 epochs after each commit interval. This would not require much change in the NFQ-SARSA(\(\lambda\)) algorithm used for this thesis, but since the cache commit interval and the number of epochs used for training are that large, the result will be quite different.

With that in mind, the NFQ-SARSA(\(\lambda\)) algorithm used in this thesis, could with slight modifications be parameterized to fill the complete spectrum between the off-line method used by Kalyanakrishnan and Stone (2007) and the on-line method used in this thesis, and I am confident that a set of parameters that can scale to the backgammon problem can be found within this spectrum.

The parameterized NFQ-SARSA(\(\lambda\)) algorithm, will however, still be very time consuming, and I think that it is important to limit this consumption. I will here give a number of methods which can reduce the execution time for the NFQ-SARSA(\(\lambda\)) algorithm, and which may help in making the algorithm scale towards larger problem sizes:

**Early stopping:** For keepaway soccer 1000 epochs of neural network training was used no matter how much the training reduced the MSE. Early stopping is a method often used when training neural networks where the training is stopped when the MSE is either below a specific limit, or it has not changed significantly for predefined number of epochs. This method would be able to reduce the number of episodes used for training the neural network by not necessarily training for 1000 epochs each time.

**Cache pruning:** The cache is set to a specific size, and the oldest elements of the cache are removed when new elements are inserted. However, some elements in the cache are more important for learning than other, and the cache can be kept smaller if only the most important elements are kept in the cache. One method of removing unimportant elements from the cache, is to simply remove the elements with the smallest neural network error, since these elements are already learned by the neural network. However, there is a risk that this pruning method will remove many beneficial elements from the cache, and only keep the strange elements that are hard to learn. For this reason I recommend combining this method with an age property, and prune elements based on a weighted average of the two. It might also be useful to make sure that all goal states are always represented by at least one sample in the cache.
8.2. SCALING THE NFQ-SARSA($\lambda$) ALGORITHM

SARSA learning: One of the most time consuming elements of NFQ-SARSA($\lambda$) is the recalculation, and especially the part where a maximum of all possible actions must be calculated. If the $\sigma$ parameter is set to 1.0, the algorithm will use SARSA learning, and only the value for the actual taken action needs to be calculated. However, using pure SARSA may prevent experience from different runs from being combined correctly and a pseudo $Q$-learning technique might be a better idea.

Pseudo $Q$-learning: If the $\sigma$ parameter is not set to 1.0, the maximum will need to be calculated for each sample before each epoch. However, since the algorithm will train the neural network with the same set of experiences in many epochs in a row, the maximum action will probably be the same for many of the samples from one epoch to the other. For this reason, a good approximation to $Q$-learning can be achieved by only recalculating the maximum action every 10 epochs or so, meaning that in the remaining 9 epochs only one action will need to be recalculated. This will especially help for a problem such as backgammon where there may be several hundred available actions.

Semi recalculation: As mentioned, the recalculation is very time consuming and the network is trained for many epochs, with the same set of experiences. To save time, it could be an idea not to recalculate the cache before each epoch and train the network for e.g. 10 epochs with the same training data, before recalculating the cache. This method will also help convergence for the neural network, since the training data will not change for each epoch.

The semi recalculation is actually a method that combines NFQ-SARSA($\lambda$) and $Q$-SARSA($\lambda$), since NFQ-SARSA($\lambda$) recalculates the $Q[s, a]$ values and $Q$-SARSA($\lambda$) does not. If the recalculation happens often, the algorithm is close to the NFQ-SARSA($\lambda$) algorithm, and if it happens rarely the algorithm is close to the $Q$-SARSA($\lambda$) algorithm.

8.2.2 Combining NFQ-SARSA($\lambda$) and $Q$-SARSA($\lambda$)

There are two central elements of the NFQ-SARSA($\lambda$) algorithm which I think could help enhance the $Q$-SARSA($\lambda$) algorithm: The recalculation, and the idea of training several times without adding new training data.

If the recalculation is added before every training epoch, then the $Q$-SARSA($\lambda$) algorithm has simply been changed into the NFQ-SARSA($\lambda$) algorithm, but recalculating every time might not always be a good idea. Partly because of time consumption and partly because the recalculation will make all data in the training set change between each epoch, thereby hindering convergence.

The simplest method for adding recalculation is the semi recalculation method, but there are also other methods. When the $Q$-SARSA($\lambda$) algorithm needs to get a value from the sliding window cache, it will sometimes find a state-action pair which is already available in the cache. The $Q[s, a]$ value for this state-action pair could be recalculated before it is returned, and the recalculation will then happen as needed, instead of the complete recalculation. Another method for introducing recalculation as needed, is to only include recalculation when candidates are installed. When it has been decided that a candidate neuron should be added to the network, the network could be trained for a number of epochs with recalculation in order to ensure that the neural network and cache are at a stable state, so that the candidate has the optimal conditions. When the candidate is fully installed, the network could be trained for a number of episodes again to ensure that the candidate is fitted nicely into the network.
CHAPTER 8. FUTURE WORK

This method also introduces the idea of training for several epochs without adding new training data. This could also be introduced in other situations where it is needed. A special situation, which I have observed during some tests, is where the MSE grows dramatically, most likely because a new policy has been observed. In these situations it might be a good idea to train the neural network for several episodes, until the MSE have been stabilized again. This training could both be done with or without recalculation.

I think that NFQ-SARSA(\(\lambda\)) and \(Q\)-SARSA(\(\lambda\)) are algorithms that are so closely related, that it would make sense to join them as one combined algorithm. This algorithm could use some of the ideas that I have proposed in this section in order to obtain a much higher quality of learning and scaling.

8.3 Learning by Example

Reinforcement learning is in many ways based on trial-and-error, and this is part of what makes the algorithms so interesting. Trial-and-error is also one of the basic learning mechanisms for humans, but another basic learning mechanism for humans is learning by example, where the child e.g. observes how the parents walk, and tries to mimic that.

Learning-by-example is the basic learning method for neural networks, where the neural network is told how to act in certain situations and it tries to mimic that. This method was used in the predecessors to TD-Gammon (Tesauro and Sejnowski, 1989), and the main weakness of this method was that it could never learn to be better than its teacher.

Learning-by-example can also be combined with reinforcement learning, and the basic idea of this approach is that the agent is shown how to solve a task, like e.g. driving a car up a mountain, and it then uses this information to create its own policy for handling the same task. This idea is not new (Atkeson and Schaal, 1997; Price, 2003), but the introduction of the NFQ and NFQ-SARSA(\(\lambda\)) algorithms introduces a whole new method of implementing this.

The NFQ-SARSA(\(\lambda\)) algorithm needs \(\langle s, a, r, s', a' \rangle\) tuples in order to learn, but unlike the \(Q\)-SARSA(\(\lambda\)) algorithm it does not need a \(Q(s, a)\) value. The NFQ-SARSA(\(\lambda\)) algorithm can use these tuples to learn a policy, and this policy is learned completely independently of the policy used when the tuples were gathered, although elements of a good policy are needed in the tuples in order to learn a good policy. In the original NFQ algorithm these elements are initially gathered by following a random policy. However, they could just as well be gathered by observing the actions taken by an expert agent. A NFQ-SARSA(\(\lambda\)) agent could e.g. play backgammon against an expert opponent, and it could add the tuples from its own moves along with the tuples from the opponents moves to the cache. I believe that this would be able to dramatically speed up training, since the agent would be shown some expert moves that would take a long time for itself to discover. The NFQ-SARSA(\(\lambda\)) algorithm will be able to benefit from both small amounts of learning-by-example and large amounts, especially since it can choose to keep the expert tuples in the cache long after it has switched back to self play.

Learning-by-example can be applied in many different situations, where it is possible for the agent to observe an expert agent, or where it is possible for an expert to remote control the agent. One such field which is very interesting, is the field of artificial intelligence in computer games. An example is the Quake III bot (Waveren, 2001), which is an AI player in the first person shooter game of Quake III. This agent could learn from being remote controlled by an expert player, and it could then elaborate on this learning, by playing on its own. If it turns out that the agent has problems with some specific strategies, the agent could be sent back
to “school” and the expert player could remote control it for a while, and show how these strategies could be executed.
Appendix A

Artificial Neural Network Basics

This chapter have in large been taken from (Nissen, 2003) and is included to give a basic introduction to ANNs.

A.1 Neural Network Theory

This section will briefly explain the theory of neural networks (NN) and artificial neural networks (ANN). For a more in depth explanation of these concepts please consult the literature; Hassoun (1995) has good coverage of most concepts of ANN and Hertz et al. (1991) describes the mathematics of ANN very thoroughly, while Anderson (1995) has a more psychological and physiological approach to NN and ANN. For the pragmatic I could recommend Tettamanzi and Tomassini (2001), which has a short and easily understandable introduction to NN and ANN.

A.1.1 Neural Networks

The human brain is a highly complicated machine capable of solving very complex problems. Although we have a good understanding of some of the basic operations that drive the brain, we are still far from understanding everything there is to know about the brain.

In order to understand ANN, you will need basic knowledge of how the internals of the brain functions. The brain is part of the central nervous system and consists of a very large NN. The NN is quite complicated, but I will only include the details needed to understand ANN, to simplify the explanation.

Figure A.1: Simplified neuron.
APPENDIX A. ARTIFICIAL NEURAL NETWORK BASICS

The NN is a network consisting of connected neurons. The center of the neuron is called the nucleus. The nucleus is connected to other nuclei by means of the dendrites and the axon. This connection is called a synaptic connection.

The neuron can fire electric pulses through its synaptic connections, which is received at the dendrites of other neurons. Figure A.1 shows what a simplified neuron looks like. When a neuron receives enough electric pulses through its dendrites, it activates and fires a pulse through its axon, which is then received by other neurons. In this way information can propagate through the NN. The synaptic connections change throughout the lifetime of a neuron and the amount of incoming pulses needed to activate a neuron (the threshold) also change. This behavior allows the NN to learn.

The human brain consists of around $10^{11}$ neurons which are highly interconnected with around $10^{15}$ connections (Tettamanzi and Tomassini, 2001). These neurons activate in parallel as an effect to internal and external sources. The brain is connected to the rest of the nervous system, which allows it to receive information by means of the five senses and also allows it to control the muscles.

A.1.2 Artificial Neural Networks

It is not possible (at the moment) to make an artificial brain, but it is possible to make simplified artificial neurons and artificial neural networks. These ANNs can be use many different topologies and can try to mimic the brain in many different ways.

ANNs are not intelligent, but they are good for recognizing patterns and making simple rules for complex problems. They also have excellent training capabilities which is why they are often used in artificial intelligence research.

ANNs are good at generalizing from a set of training data. If an ANN is e.g. shown examples of many different animals, along with information about whether they are mammals or not, it will be able to tell if new animals are mammals or not, even though they were not part of the initial set of examples. This is a very desirable feature of ANNs, because you do not need to know the characteristics defining a mammal, the ANN will find out by itself.

The Artificial Neuron

A single artificial neuron can be implemented in many different ways. The general mathematic definition is as showed in equation A.1.1.

\[
y(x) = g \left( \sum_{i=0}^{n} w_i x_i \right)
\]  

(A.1.1)

$x$ is a neuron with $n$ input dendrites ($x_0 \ldots x_n$) and one output axon $y(x)$ and where ($w_0 \ldots w_n$) are weights determining how much the inputs should be weighted.

$g$ is an activation function that weights how powerful the output (if any) should be from the neuron, based on the sum of the input. If the artificial neuron should mimic a real neuron, the activation function $g$ should be a simple threshold function returning 0 or 1. This is however, not the way artificial neurons are usually implemented. For many different reasons it is better to have a smooth (preferably differentiable) activation function. The output from most activation functions is either between 0 and 1, or between -1 and 1, depending on which activation function is used. There are, however, exceptions like the identity function, which does not have these limitations. The inputs and the weights are not restricted in the same
way and can in principle be between $-\infty$ and $+\infty$, but they are often small values centered around zero. The artificial neuron is also illustrated in figure A.2.

In the figure of the real neuron (figure A.1), the weights are not illustrated, but they are implicitly given by the number of pulses a neuron sends out, the strength of the pulses and how closely connected the neurons are.

As mentioned earlier there are many different activation functions, some of the most commonly used are threshold (A.1.2), sigmoid (A.1.3) and hyperbolic tangent (A.1.4), where $t$ is a value that pushes the center of the activation function away from zero and $s$ is a steepness parameter.

$$g(x) = \begin{cases} 1 & \text{if } x + t > 0 \\ 0 & \text{if } x + t \leq 0 \end{cases}$$ (A.1.2)

$$g(x) = \frac{1}{1 + e^{-2s(x+t)}}$$ (A.1.3)

$$g(x) = \tanh(s(x+t)) = \frac{\sinh(s(x+t))}{\cosh(s(x+t))} = \frac{e^{s(x+t)} - e^{-s(x+t)}}{e^{s(x+t)} + e^{-s(x+t)}} = \frac{e^{2s(x+t)} - 1}{e^{2s(x+t)} + 1}$$ (A.1.4)

Sigmoid and hyperbolic tangent are both smooth differentiable functions, with very similar graphs, the only major difference is that hyperbolic tangent has output that ranges from -1 to 1 and sigmoid has output that ranges from 0 to 1. A graph of a sigmoid function is given in figure A.3.

The $t$ parameter in an artificial neuron can be seen as the amount of incoming pulses needed to activate a real neuron. This parameter, together with the weights, are the parameters adjusted when the neuron learns.
The Artificial Neural Network

The most common kind of ANN is the multilayer feedforward ANN, please see Hassoun (1995) for a description of other ANN kinds. In a multilayer feedforward ANN, the neurons are ordered in layers, starting with an input layer and ending with an output layer. Between these two layers are a number of hidden layers. Connections in these kinds of network only go forward from one layer to the next.

Multilayer feedforward ANNs have two different phases: A training phase (sometimes also referred to as the learning phase) and an execution phase. In the training phase the ANN is trained to return a specific output when given a specific input, this is done by continuous training on a set of training data. In the execution phase the ANN returns outputs on the basis of inputs.

The execution of a feedforward ANN functions as follows: An input is presented to the input layer, the input is propagated through all the layers (using equation A.1.1) until it reaches the output layer, where the output is returned. In a feedforward ANN an input can easily be propagated through the network and evaluated to an output. It is more difficult to compute a clear output from a network where connections are allowed in all directions (like in the brain), since this will create loops. There are ways of dealing with these loops in recurrent networks, and the loops can be used to code time dependencies as described by (Hassoun 1995 p. 271). However, for problems that are not time-dependent feedforward networks are usually a better choice.

Two different kinds of parameters can be adjusted during the training of an ANN, the weights and the $t$ value in the activation functions. This is impractical and it would be easier if only one of the parameters should be adjusted. To cope with this problem a bias neuron is invented. The bias neuron lies in one layer, is connected to all the neurons in the next layer, but none in the previous layer and it always emits 1. Since the bias neuron emits 1 the weights, connected to the bias neuron, are added directly to the combined sum of the other weights (equation A.1.1), just like the $t$ value in the activation functions. A modified equation for the neuron, where the weight for the bias neuron is represented as $w_{n+1}$, is shown in equation A.1.5.

\[
y(x) = g \left( w_{n+1} + \sum_{i=0}^{n} w_i x_i \right) \tag{A.1.5}
\]
Adding the bias neuron allows us to remove the $t$ value from the activation function, only leaving the weights to be adjusted, when the ANN is being trained. A modified version of the sigmoid function is shown in equation A.1.6.

$$g(x) = \frac{1}{1 + e^{-2sx}}$$  \hspace{1cm} (A.1.6)

The $t$ value cannot be removed without adding a bias neuron, since this would result in a zero output from the sum function if all inputs were zero, regardless of the values of the weights. Some ANN libraries do however remove the $t$ value without adding bias neurons, counting on the subsequent layers to get the right results. An ANN with added bias neurons is shown in figure A.5.

![Figure A.5: A fully connected multilayer feedforward network with one hidden layer and bias neurons.](image)

**Running Time of Executing an ANN**

When executing an ANN, equation A.1.5 needs to be calculated for each neuron which is not an input or bias neuron. This means that one multiplication and one addition have to be executed for each connection (including the connections from the bias neurons), besides that one call to the activation function need also be made, for each neuron that is not an input or bias neuron. This gives the following running time:

$$T(n) = cA + (n - n_i)G$$  \hspace{1cm} (A.1.7)

Where $c$ is the number of connections, $n$ is the total number of neurons, $n_i$ is the number of input and bias neurons, $A$ is the cost of multiplying the weight with the input and adding it to the sum, $G$ is the cost of the activation function and $T$ is the total cost.

If the ANN is fully connected, $l$ is the number of layers and $n_l$ is the number of neurons in each layer (not counting the bias neuron), this equation can be rewritten to:

$$T(n) = (l - 1)(n_i^2 + n_l)A + (l - 1)n_lG$$  \hspace{1cm} (A.1.8)

This equation shows that the total cost is dominated by $A$ in a fully connected ANN. This means that if the execution of a fully connected ANN should be optimized, $A$ and retrieval of the information needed to compute $A$ need to be optimized.
A.1.3 Training an ANN

When training an ANN with a set of input and output data, the weights in the ANN should be adjusted to make the ANN give the same outputs as seen in the training data. However, it is not desirable to make the ANN too specific, making it give precise results for the training data, but incorrect results for all other data. When this happens, we say that the ANN has been over-fitted.

The training process can be seen as an optimization problem, where the mean square error of the entire set of training data should be minimized. This problem can be solved in many different ways, ranging from standard optimization heuristics like simulated annealing, through more special optimization techniques like genetic algorithms to specialized gradient descent algorithms like back-propagation.

The most used algorithm is the back-propagation algorithm (see section A.1.3), but this algorithm has some limitations concerning the extent of adjustment to the weights in each iteration. This problem has been solved in more advanced algorithms like RPROP (Riedmiller and Braun, 1993) and Quickprop (Fahlman, 1988), which is described in further detail in chapter 2.

The Back-Propagation Algorithm

The back-propagation algorithm works in much the same way as the name suggests: After propagating an input through the network, the error is calculated and the error is propagated back through the network while the weights are adjusted in order to make the error smaller. Here I refer to fully connected ANNs, but the theory is the same for sparsely connected ANNs.

Although the mean square error for all the training data should be minimized, the most efficient way of doing this with the back-propagation algorithm, is to train on data sequentially one input at a time, instead of training on the combined data. However, this means that the order the data is given in is of importance, but it also provides a very efficient way of avoiding getting stuck in a local minima.

I will now explain the back-propagation algorithm, in sufficient details to allow an implementation from this explanation. First the input is propagated through the ANN to the output. After this the error $e_k$ on a single output neuron $k$ can be calculated as:

$$e_k = d_k - y_k$$  \hspace{1cm} (A.1.9)

Where $y_k$ is the calculated output and $d_k$ is the desired output of neuron $k$. This error value is used to calculate a $\delta_k$ value, which is again used for adjusting the weights. The $\delta_k$ value is calculated by:

$$\delta_k = e_k \cdot g'(y_k)$$  \hspace{1cm} (A.1.10)

Where $g'$ is the derived activation function. The need for calculating the derived activation function was why I expressed the need for a differentiable activation function in section A.1.2.

When the $\delta_k$ value is calculated, the $\delta_j$ values for preceding layers can be calculated. The $\delta_j$ values of the previous layer is calculated from the $\delta_k$ values of this layer. By the following equation:

$$\delta_j = \eta \cdot g'(y_j) \sum_{k=0}^{K} \delta_k w_{jk}$$  \hspace{1cm} (A.1.11)

Where $K$ is the number of neurons in this layer and $\eta$ is the learning rate parameter, which determines how much the weight should be adjusted. The more advanced gradient descent algorithms do not use a learning rate, but a set of more
advanced parameters that make a more qualified guess to how much the weight should be adjusted.

Using these $\delta$ values, the $\Delta w$ values that the weights should be adjusted by, can be calculated by:

$$\Delta w_{jk} = \delta_j y_k$$  \hfill (A.1.12)

The $\Delta w_{jk}$ value is used to adjust the weight $w_{jk}$, by $w_{jk} = w_{jk} + \Delta w_{jk}$ and the back-propagation algorithm moves on to the next input and adjusts the weights according to the output. This process goes on until a certain stop criteria is reached. The stop criteria is typically determined by measuring the mean square error of the training data while training with the data, when this mean square error reaches a certain limit, the training is stopped. More advanced stopping criteria involving both training and testing data are also used.

In this section I have briefly discussed the mathematics of the back-propagation algorithm and I have left out details unnecessary for implementing the algorithm. I will refer to Hassoun (1995) and Hertz et al. (1991) for more detailed explanation of the theory behind and the mathematics of this algorithm.
Appendix B

Backgammon Input Representation

Since backgammon can use board positions as after-states the agent is presented with a number of possible after-states, and it will have to determine which of them that is most profitable. A simple heuristic for this can be implemented by looking at the pip count. The pip count is a number which indicates the total number of places that the players checkers will have to move, before they are all borne off the board. When the game begins, the pip count is 167 for both players. A simple heuristic for comparing board position will simply compare the pip count for both players and choose the board position which is most profitable. However, looking at the pip count is not sufficient to provide a good heuristic. A better heuristic will also consider the number of places which is only occupied by one checker (known as a blot), since these pose a risk of being hit and the heuristic will also need to know when all checkers are in the home board, since it is not allowed to bear checkers off before all checkers are home. A number of other indicators can be suggested, which must also be considered by a good heuristic. I would suggest a strategy for blocking the opponent by placing two or more checkers on places between the opponents checkers and his home, and I am sure that an expert player can suggest many more indicators which must be considered.

All of these indicators must be considered when selecting the input representation for the neural network. However when the problem is solved by reinforcement learning with a neural network, all of these indicators can be learned by the neural network, and it is not absolutely necessary to represent them in the input encoding. Two rules of thumb exist for representing input to a neural network which can be used in this situation to determine how a board position should be represented, and which indicators that should be included in the representation:

- The input representation should be as simple as possible, and should not include any unnecessary information.
- All the knowledge, that the person designing the input representation has about the problem, should be included in the representation.

These rules of thumb indicate that all relevant information that can be gathered from the board, should included in the input representation, but that no information that is not relevant should be included. The question of relevant information is hard to determine, but some of the most important informations are:

- The pip count
• Indication of which positions that has exactly one checker, since these positions may be hit.

• Indicating of which positions that have more than one checker on them, since these positions block the opponent.

The representation used by Bellemare et al. (2004) does not include the pip count, but it does have a very thorough representation of each position, which also include information about which positions that have more than one checker. The representation represent each position as four inputs, the first three inputs are binary indicators that are set to one if there are respectively 1 or more, 2 or more and 3 or more, checkers on the position. The fourth input is a decimal number which is only used if there are more than three checkers on the position and the number is the number of checkers more than three divided by two. There are used four inputs for each of the players, which means that there are used a total of eight inputs for each position, and since there are 24 positions in backgammon there is a total of 192 inputs used to represent the 24 positions. Besides this one input is used for each player to represent the number of checkers on the bar and the number of checkers that have already been borne off, which gives a total of 196 inputs.

This representation is very close to the representation used by the first version of TD-Gammon as described by Sutton and Barto (1998), with the only modification that no information is included about who’s turn it is. The turn information is a redundant information, since it will always be the players turn when the board is presented to the player, and since the board will be mirrored, so that it will look the same no matter if the agent is playing white or red. Although this has been documented by Sutton and Barto (1998) as the representation used by TD-Gammon, this claim has never been verified by Tesauro, who has only described his representation in vague terms after his success with TD-Gammon, and I have serious doubts that this was the actual representation used. This representation has one major disadvantage in the fact that you will have to look at two inputs in order to identify the relative simply feature of a position with exactly one checker on (a blot). In Tesauro and Sejnowski (1989) and Tesauro (1992) which precedes TD-Gammon, Tesauro suggests two other representations, which both use four inputs to represent a position, and which both use the first three inputs to represent 1, 2 and 3 checkers and where the fourth input represents 4 and more checkers, just like the representation described by Sutton and Barto (1998). The difference lies in how exactly the first three inputs are modelled. The first representation set the inputs to one if the position has exactly 1, 2 or 3 checkers respectively. This representation has the advantage, that you only have to look at the first input in order to detect a blot, but the disadvantage is that you have to look at all of the remaining three inputs to detect a blockade. The second representation set the first and third input to one if the position have 1 or 3 checkers respectively, and set the second input to one if there are 2 or more checkers in the position. This representation has the advantage, that you only have to look at one input to detect a blot and at one input to detect a blockade. Figure B.1 shows how the three representations can be used to represent 1, 2, 3 and 4 checkers, and I feel confident that the second representation is superior to both the first representation and the representation described by Sutton and Barto (1998), and since this representation was described by Tesauro before TD-Gammon, I have a hard time believing that he would choose a weaker representation for the first version of TD-Gammon. For later versions (Tesauro, 1995) used a set of hand-crafted features, in addition to the raw board encoding. The actual features used by later versions of TD-Gammon is not known, but the earlier publication by Tesauro and Sejnowski (1989) mentions eight features that could be included in the input representation. These features are: 1) pip count, 2) degree of contact, 3) points occupied in own inner board, 4) points
occupied in opponent’s inner board, 5) number of men in opponent’s inner board, 6) presence of a “prime” formation (a blockade of 6 points, which is impossible to clear), 7) the probability that a blot can be hit and 8) the strength of a blockade.

![Diagram showing checker configurations](image)

Figure B.1: Demonstration of how the three representations will represent 1, 2, 3 and 4 checkers. It can be seen that when the first neuron is set, this represents a blot in Tesauro (1989) and Tesauro (1992), and when the second neuron is set, this represents a blockade in Tesauro (1992) and Sutton (1998), so only Tesauro (1992) can detect both of these two situations by looking at single neurons.

Features like these require knowledge of the game, and can be designed in cooperation with an expert backgammon player. However, as mentioned earlier in this section the neural network is a general purpose function approximator, and should be able to extract these features on its own, if they have any significant impact on final result. The neural network would probably be able to extract some of these features on its own, but this would waste space in the neural network, that could be used to extract more high level features, and learning the features would require a significant amount of time.

In order to help the neural network achieve good performance, it is of great importance to design good features. While doing so, it must be kept in mind that only relevant features should be added and that the features should only include information that can be calculated directly from the board and no speculative information, which indicates if the feature is positive or negative for the final result. The speculation should be left to the reinforcement learning and neural network, so that it can be trained to know in which situations a feature is beneficial for the player and in which situations it is not.
Appendix C

Tuning of $Q$-SARSA($\lambda$) for Backgammon

When tuning the cascading $Q$-SARSA($\lambda$) algorithm for the backgammon problem a number of different modifications to the algorithm was tried which did not improve the performance. The results from these modifications are not included in chapter 6 because they do not improve the performance. However, it is interesting and important to know why the modifications did not succeed in producing better results and this is documented in this appendix. The tests in this section are made on the basis of the tuned parameters for tuning session B in section 6.8.5 on page 144.

C.1 Using a Larger Cache for Candidates

Tuning session B allowed for a larger cache to be used when the candidates are trained. Figure C.1 show the results of this test, and with a cache size of 7500 and 20,000 there is a degeneracy, but for the remaining cache sizes the performance is only slightly worse than when a cache size of 4000 is used for both the neural network training and the candidate training. This suggest that although there is generally no harm in using a different cache size when training the candidates, there is no clear advantage of this approach.

C.2 Tabu Search

There was hope that tabu search might help boost the performance. However, as can be seen in figure C.2 tabu search actually decreases performance, and the tendency is clear, as the size of the tabu list increases, the performance decreases. The tabu search which is used in this thesis, is the version where tabu search is only used for the explorative actions, and where it is not allowed to select an explorative action which is in the tabu list, unless no other explorative action is available. For the backgammon problem actions are represented as board positions, so if the tabu size is 100, this means that the last 100 board positions, which have been selected as explorative actions is in the tabu list. These 100 board positions are illegal when selecting a new explorative action, while they are perfectly legal when selecting an exploitive action. With an $\epsilon$ value of 0.01 this means that the same explorative action may not be selected for approximately 10,000 steps (or 100 games). For the backgammon problem which have many different board positions this should not have a large effect, except in the opening position, where the same board positions will often be possible. The tabu search should ensure that all the possible board
APPENDIX C. TUNING OF $Q$-SARSA($\lambda$) FOR BACKGAMMON

Figure C.1: Average off-line win percentage, for the cascading $Q$-SARSA($\lambda$) trained backgammon problem with a larger cache, where the normal cache size is fixed at 4000, but where the cache used to train the candidates are allowed to have a size of 5000, 7500, 10,000 and 20,000. The win percentage is measured after 10,000 games, as an average of 1000 games.

Initially this does not seem to make much sense, but the tendency is clear, and the tendency consists throughout the tuning iterations. A closer look at the after-state aspect of the backgammon problem, does however, reveal a problem which positions are tested, which should ensure better learning. However, it seems like tabu search actually hinders learning by not allowing the same explorative board positions to be selected several times, and hence not allowing learning to occur for these positions.

Figure C.2: Average off-line win percentage, for the cascading $Q$-SARSA($\lambda$) trained backgammon problem with a larger cache, for variations of the tabu size parameter with the remaining parameters fixed at their tuned values. The win percentage is measured after 10,000 games, as an average of 1000 games.
means that tabu search is not a good idea when after-states are used.

Tabu search functions like this without after-states: In state $s$ there are three possible actions $a$, $b$ and $c$. Action $a$ is the exploitive action, so if an explorative action should be selected action $b$ and $c$ may be selected. If action $b$ is selected, then action $c$ will be selected the next time an explorative action should be selected in state $s$, which ensures that the state-action space is explored more uniformly. When after-states are used, the same scenario can be seen, but here the actions are just represented as after-states which are board positions, and the states are as such not represented, since the state information is included in the board position. In this case, when board position $b$ is selected, this board position is not only made tabu from state $s$, it is made tabu from all states, since states are not represented directly. This poses a problem, since it actually means that if $b$ is also an after-state of state $t$, then it is not allowed to select this after-state as an explorative action, and the agent may be forced to select board position $d$. The effect of this is, that when the agent is in state $u$, it is more likely to select an explorative after-state which can only be reached from $u$, than selecting an after-state which can be reached from many different states. Which actually means that tabu search forces the agent to use time exploring after-states which does not occur that often, instead of exploring after-states that occur often. This is in contradiction with one of the main ideas of reinforcement learning, which states that more effort should be used on frequently occurring situations than on rarely occurring situations. So for reinforcement learning which uses after-states, tabu search actually assures that more parts of the state space is visited, but it does so at the expense of some of the more frequently occurring states, which is generally not desired.
APPENDIX C. TUNING OF Q-SARSA($\lambda$) FOR BACKGAMMON
Appendix D

Comparing $\text{NFQ-SARSA}(\lambda)$ to NFQ

The NFQ-SARSA($\lambda$) algorithm adds several enhancements to the NFQ algorithm. This appendix will discuss how these enhancements influence the performance of the algorithm.

If the results of the cascading NFQ-SARSA($\lambda$) algorithm for the mountain car and cart pole problems should be compared directly to that of the original NFQ algorithm at the NIPS conference, the NFQ algorithm would be declared the winner because it was able to achieve better performance for the mountain car problem in fewer episodes and because it was able to achieve comparable performance for the cart pole problem in fewer episodes. However, the comparison cannot be made as simple. The NFQ algorithm at the NIPS conference used several heuristics to increase the performance, and the cache was handled quite differently than the cache for the NFQ-SARSA($\lambda$) algorithm.

Another way of comparing the NFQ algorithm to the NFQ-SARSA($\lambda$) is by looking at some of the extra parameters that was added to the algorithm, since all of these parameters could be set, so as to make the NFQ-SARSA($\lambda$) into the NFQ algorithm. If this comparison should be made completely unbiased, all of the parameters should be tested one by one, by adding them to the basic NFQ algorithm and measure the performance of this addition. However, this is a very time consuming activity, and I will make a simpler comparison, by inspecting the graphs for the performance for variations of the parameters on the mountain car and cart pole problems, leaving out the backgammon problem, due to substandard performance.

The parameters which are added to the NFQ-SARSA($\lambda$) algorithm are the $\alpha$, $\lambda$, $\sigma$, cache size and cache commit interval parameters which can be inspected in figure 6.28 and 6.29 on page 132 for the mountain car problem, and in figure 6.32 and 6.30 on page 133 for the cart pole problem.

Replacing eligibility traces introduce the $\lambda$ parameter, and if it is set to zero it will be the same as not introducing it. For the mountain car problem, it is difficult to see a clear trend for this parameter, but it is clear for the cart pole problem where it should be as high as possible, so it would seem that this is a valuable addition.

The $Q$-SARSA algorithm introduces the $\sigma$ parameter, which will reduce to $Q$-learning if it is set to zero, which also means that the NFQ-SARSA($\lambda$) algorithm will reduce to the NFQ($\lambda$) algorithm if $\sigma$ is zero. Although the trend is not as clear, neither of the problems perform very well when the $\sigma$ parameter is zero, so it would also seem that this parameter is beneficial.

The $\alpha$ parameter was originally not needed for the NFQ algorithm, since the
complete training set was used for training all the time. The \( \alpha \) parameter was introduced as a consequence of the fact that the complete training set could not be used for training all the time, and that a cache size had been introduced. If the \( \alpha \) parameter is one, it would be the same as not having introduced it. For both problems a large \( \alpha \) parameter close to one is the optimal, but one is not the optimal since it will allow earlier learned values to be forgotten. I believe that the \( \alpha \) parameter is essential, if there should be a fixed upper bound on the cache size. This raises the question of whether it is a good idea to have a fixed upper bound on the cache size? When only looking at the test results, it would seem that it is, since the best performing sizes are 500 and 750 for the mountain car and cart pole problems respectively. However, since the cache is not used in exactly the same way as the original NFQ algorithm, it is hard to make this judgement directly. I do, however, believe that a upper bound on the cache size is needed for more advanced problems like e.g. backgammon, so it is a valuable addition. The cache commit interval is valuable in the NFQ-SARSA(\( \lambda \)) algorithm, but although this particular parameter is added by the NFQ-SARSA(\( \lambda \)) algorithm, it could just as well have been included in the original NFQ algorithm, since it does not specifically specify how often the neural network should be trained.

It is clear that the enhancements to the NFQ algorithm provided by the NFQ-SARSA(\( \lambda \)) algorithm, are valuable in the context where they have been tested here, but it is uncertain if the enhancements are generally valuable for the original NFQ algorithm. In order to determine exactly how valuable the enhancements are for the original NFQ algorithm, more benchmarks can be made that try to more closely mimic the behavior of the original NFQ algorithm. However, I believe that these tests will show the same trends that have been seen in the tests conducted for this thesis; that generally the enhancements are beneficial. And even if some of the enhancements are not beneficial, they can easily be turned off.
Appendix E

Using the Implementation

This appendix provides a short introduction to how most of the benchmarks in this thesis can be replicated. The source code for replicating the benchmarks are available from src.tgz file located on the CD-ROM that accompanies the thesis, and can also be downloaded from: http://leenissen.dk/rl/

The implementation have been developed on a Linux machine, and although there is nothing which should prevent the implementation from compiling under other operating systems, this have not been tested and the benchmarks rely heavily on Linux tools. For the benchmarks there will be pre-compiled executables available, which should be able to run on many Linux systems, so it is recommended that the pre-compiled executables are tried before compiling anything. However, the neural network benchmarks require that the FANN library is installed, so that should be compiled and installed first.

E.1 The Neural Network Implementation

The neural network implementation used for this thesis is the FANN library version 2.1.0beta. However, in order to execute the benchmarks minor modifications have been made, so the implementation provided on the CD-ROM should be used.

E.1.1 Compiling the FANN Library

In the src/RL/fann/ directory, execute the following commands:

```bash
./configure
make
```

Followed by a `make install` executed as `root`.

E.1.2 Compiling the Benchmarks

The benchmarks are located in the src/RL/fann/benchmarks/ directory. If the benchmarks should include the lwmm library and the jneural library, then these libraries should be downloaded as described in the `Makefile`. If they should not be included, two lines should be uncommented in the beginning of the `quality.cc` and `performance.cc` files (instructions for uncommenting located inside files). After this have been done, simply execute the `make` command.
E.1.3 Executing the Benchmarks

In the `src/RL/fann/benchmarks/` directory, execute the following commands:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/usr/local/lib
bash benchmark.sh
```

The benchmarks will create graphs that will be visible in the `allplots.ps` file, and summary data will be written to the `summary.txt` file, which can be used to generate LaTeX tables by means of the `summary2tex.pl` file (minor editing of paths in file needed).

The benchmarks will not be 100% the same as the benchmarks documented in this thesis, since the random number generator is not seeded with a fixed value.

E.2 The Reinforcement Learning Implementation

The reinforcement learning implementation uses a slightly modified version of the RL-Glue library, which is included in the download. In addition to this, the backgammon implementation is in Java, so a Java runtime environment is needed to execute this.

E.2.1 Compiling the Benchmarks

The blackjack, mountain car, cart pole and backgammon implementation can be compiled by executing the following commands in the `src/RL/` directory:

```
make -f makefile.bj clean
make -f makefile.bj
make -f makefile.car clean
make -f makefile.car
make -f makefile.cart clean
make -f makefile.cart
make -f makefile.bg clean
make -f makefile.bg pipes
```

E.2.2 Executing the Benchmarks

Shell scripts have been made available for executing most of the benchmarks from the thesis, and they can be executed as described in figure E.1.

Since the benchmarks have been executed with a fixed random number initialization, it should be possible to recreate the benchmarks. However, since random numbers are used for several different parts of the implementation, it have been seen that minor changes to the program, could change the result of the benchmarks. It is e.g. possible to influence the random numbers, by changing how often the agent is saved to the disk. For this reason it will not be possible to recreate all of the benchmarks exactly as they are presented in this thesis.

Command Line Parameters

If new benchmarks should be executed, looking in some of the benchmark shell files is a good place to get started. However, the executables for the individual problems can also be executed from the command line, by using the command line
E.2. THE REINFORCEMENT LEARNING IMPLEMENTATION

<table>
<thead>
<tr>
<th>Shell file</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench_bj_tabular.sh</td>
<td>Tabular $Q$-SARSA($\lambda$) blackjack</td>
</tr>
<tr>
<td>bench_car_incr.sh</td>
<td>Incremental $Q$-SARSA mountain car</td>
</tr>
<tr>
<td>bench_cart_incr.sh</td>
<td>Incremental $Q$-SARSA cart pole</td>
</tr>
<tr>
<td>bench_car_batch.sh</td>
<td>Batch $Q$-SARSA($\lambda$) mountain car</td>
</tr>
<tr>
<td>bench_cart_batch.sh</td>
<td>Batch $Q$-SARSA($\lambda$) cart pole</td>
</tr>
<tr>
<td>bench_car_cascade.sh</td>
<td>Cascade $Q$-SARSA($\lambda$) mountain car</td>
</tr>
<tr>
<td>bench_cart_cascade.sh</td>
<td>Cascade $Q$-SARSA($\lambda$) cart pole</td>
</tr>
<tr>
<td>bench_car_cascade_nfq.sh</td>
<td>Cascade NFQ-$Q$-SARSA($\lambda$) mountain car</td>
</tr>
<tr>
<td>bench_cart_cascade_nfq.sh</td>
<td>Cascade NFQ-$Q$-SARSA($\lambda$) cart pole</td>
</tr>
<tr>
<td>bench_bg_cascade_small_cache.sh</td>
<td>Cascade $Q$-SARSA($\lambda$) backgammon (larger cache)</td>
</tr>
<tr>
<td>bench_bg_cascade.sh</td>
<td>Cascade $Q$-SARSA($\lambda$) backgammon</td>
</tr>
<tr>
<td>bench_bg_cascade_nfq.sh</td>
<td>Cascade NFQ-$Q$-SARSA($\lambda$) backgammon (larger cache)</td>
</tr>
</tbody>
</table>

Figure E.1: Shell files for the reinforcement learning tests.

parameters in figure E.2. Different executable files are available for the individual problems, and bj, car and cart are the blackjack, mountain car and cart pole problems. The backgammon problem is a bit more complex, since the Java and C++ communicates through sockets. However, some shell scripts have been made that take the exact same parameters as the executables for the other problems. The shell scripts are runbgcompetitive.sh, runbgrandom.sh and runbgheuristic.sh which will make the agent play against itself, the random agent or the heuristic agent respectively.
### Parameter (Default) | Description
--- | ---
-q (Not set) | Quiet (don’t print verbose output)
-te total-episodes (0) | Total episodes
-on online-episodes (total-episodes) | Episodes of on-line training
-off offline-episodes (total-episodes) | Episodes before switching to off-line training
-r num-runs (1) | Number of runs
-f agent-file (Not set) | Earlier saved agent to load
-select selection-strategy (1) | 1 = egreedy, 2 = maxBoltzmann
-a alpha (0.2) | \( \alpha \) parameter
-aa alpha-anneal-factor (1) | Annealing parameter for \( \alpha \)
-e epsilon (0.001) | \( \epsilon \) parameter
-ea epsilon-anneal-factor (1) | Annealing parameter for \( \epsilon \)
-s sigma (0.7) | \( \sigma \) parameter
-g gamma (0.2) | \( \gamma \) parameter
-l lambda (0.9) | \( \lambda \) parameter
-b boltzmann-temp (100) | Boltzmann temperature for max boltzmann
-ba boltzmann-anneal-factor (0.999) | Anneal parameter for boltzmann temperature
-rehearse (Not set) | Use rehearsing
-tabu tabu-size (0) | Size of tabu list
-nocache (Not set) | Do not use sliding window cache
-nfq (Not set) | Use NFQ learning
-cachesize size cand-size (2000 2000) | Size of cache used to train neural network and candidates
-cc cache-commit-interval (10) | Cache commit interval
-epoch epochs-per-commit (1) | Number of epochs of training to use for each commit
-initial initial-value (0) | Value to return before neural network have been trained, and to fill in empty places when using the tabular solution.
-la function-approximator (batch) | fann, batch, cascade, tabular
-whole-ann (Not set) | Use whole ANN training
-no-whole-ann (Set) | Do not use whole ANN training
-desired-error desired-error (0.0001) | When MSE below this value, do not add candidates
-cascade-change-fraction fraction (0.05) | Cascade change fraction
-min-epochs min-cascade-epochs (500) | Min epochs between adding candidate neurons
-max-epochs max-cascade-epochs (5000) | Max epochs between adding candidate neurons
-s-epochs stagnation-epochs (15) | Cascade patience parameter
-separate-fa (Not set) | Agents should use separate function approximator
-bf benchmark-file (Not set) | File to write benchmark values to
-bv benchmark-value (0) | Parameter value to be written to benchmark file
-timeout minutes (120) | Minutes before a timeout (0 means unlimited)
-fann-layers num-layers layer1 ... (3 196 40 1) | Number of layers for the ANN and size of the individual layers
-seed (Not set) | Seed the random number generator with a random number

Figure E.2: Command line parameters for the reinforcement learning implementation. All parameters are optional, and have reasonable default values, but if the neural network is used it is important that the input layer is set to the correct size, and if the -f parameter is set all agent and neural network parameters will be loaded from the file instead of from the command line.
Appendix F

ANN Benchmark Test Results

This appendix shows all of the graphs and tables from the benchmark results. The results are discussed in section 3.4.
APPENDIX F. ANN BENCHMARK TEST RESULTS

Table F.1: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the abelone problem.
Figure F.2: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the bank32fm problem.

Table F.2: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the bank32fm problem.
Figure F.3: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the bank32nh problem.

Table F.3: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the bank32nh problem.
Figure F.4: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the kin32fm problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th></th>
<th></th>
<th>Best Test</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.00002408</td>
<td>1</td>
<td>0.00</td>
<td>0.00225593</td>
<td>2</td>
<td>29.41</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00024834</td>
<td>2</td>
<td>10.20</td>
<td>0.00197465</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00160523</td>
<td>7</td>
<td>71.91</td>
<td>0.00258361</td>
<td>8</td>
<td>63.67</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00161189</td>
<td>8</td>
<td>72.21</td>
<td>0.00274720</td>
<td>9</td>
<td>80.78</td>
</tr>
<tr>
<td>iRPROP</td>
<td>0.00155427</td>
<td>6</td>
<td>69.59</td>
<td>0.00258335</td>
<td>7</td>
<td>63.65</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.00222288</td>
<td>10</td>
<td>100.00</td>
<td>0.00293104</td>
<td>10</td>
<td>100.00</td>
</tr>
<tr>
<td>Batch</td>
<td>0.00181264</td>
<td>9</td>
<td>81.34</td>
<td>0.00233173</td>
<td>4</td>
<td>37.34</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.00099644</td>
<td>5</td>
<td>44.22</td>
<td>0.00257839</td>
<td>6</td>
<td>63.13</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00091385</td>
<td>3</td>
<td>40.47</td>
<td>0.00240907</td>
<td>5</td>
<td>45.42</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.00092954</td>
<td>4</td>
<td>41.18</td>
<td>0.00228342</td>
<td>3</td>
<td>32.29</td>
</tr>
</tbody>
</table>

Table F.4: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the kin32fm problem.
### Table F.5: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the census-house problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th></th>
<th></th>
<th>Best Test</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.00539214</td>
<td>1</td>
<td>0.00</td>
<td>0.00753738</td>
<td>2</td>
<td>12.16</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00702730</td>
<td>2</td>
<td>40.92</td>
<td>0.00724495</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00778354</td>
<td>4</td>
<td>59.84</td>
<td>0.00792150</td>
<td>4</td>
<td>28.14</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00779252</td>
<td>5</td>
<td>60.07</td>
<td>0.00795321</td>
<td>5</td>
<td>29.46</td>
</tr>
<tr>
<td>iRPROP</td>
<td>0.00754000</td>
<td>3</td>
<td>53.75</td>
<td>0.00770007</td>
<td>3</td>
<td>18.93</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.00938843</td>
<td>10</td>
<td>100.00</td>
<td>0.00964913</td>
<td>10</td>
<td>100.00</td>
</tr>
<tr>
<td>Batch</td>
<td>0.00938589</td>
<td>9</td>
<td>99.94</td>
<td>0.00964589</td>
<td>9</td>
<td>99.87</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.00849471</td>
<td>7</td>
<td>77.64</td>
<td>0.00871677</td>
<td>6</td>
<td>61.22</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00846954</td>
<td>6</td>
<td>77.01</td>
<td>0.00873276</td>
<td>7</td>
<td>61.88</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.00932607</td>
<td>8</td>
<td>98.44</td>
<td>0.00958921</td>
<td>8</td>
<td>97.51</td>
</tr>
</tbody>
</table>

Figure F.5: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the abalone census-house problem.
Figure F.6: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the building problem.

Table F.6: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the building problem.
Figure F.7: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the diabetes problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th></th>
<th></th>
<th>Best Test</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.00000010</td>
<td>1</td>
<td>0.00</td>
<td>0.17026357</td>
<td>6</td>
<td>30.38</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00000013</td>
<td>2</td>
<td>0.00</td>
<td>0.17392187</td>
<td>7</td>
<td>43.17</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00303323</td>
<td>3</td>
<td>2.97</td>
<td>0.16882854</td>
<td>5</td>
<td>25.36</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00381572</td>
<td>4</td>
<td>3.74</td>
<td>0.16812782</td>
<td>4</td>
<td>22.91</td>
</tr>
<tr>
<td>iRPROP</td>
<td>0.10088966</td>
<td>9</td>
<td>98.81</td>
<td>0.16217817</td>
<td>2</td>
<td>2.14</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.09837770</td>
<td>8</td>
<td>96.43</td>
<td>0.16286818</td>
<td>3</td>
<td>4.53</td>
</tr>
<tr>
<td>Batch</td>
<td>0.09142332</td>
<td>6</td>
<td>89.61</td>
<td>0.16157367</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.10201881</td>
<td>10</td>
<td>100.00</td>
<td>0.19017997</td>
<td>10</td>
<td>100.00</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.09496779</td>
<td>7</td>
<td>93.09</td>
<td>0.18498123</td>
<td>9</td>
<td>81.83</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.08819902</td>
<td>5</td>
<td>86.45</td>
<td>0.18059111</td>
<td>8</td>
<td>66.48</td>
</tr>
</tbody>
</table>

Table F.7: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the diabetes problem.
Figure F.8: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the gene problem.

Table F.8: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the gene problem.
APPENDIX F. ANN BENCHMARK TEST RESULTS

Figure F.9: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the mushroom problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td></td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.00000223</td>
<td>6</td>
<td>0.20</td>
<td>0.00000309</td>
<td>5</td>
<td>0.26</td>
<td></td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00000140</td>
<td>4</td>
<td>0.12</td>
<td>0.00000474</td>
<td>6</td>
<td>0.40</td>
<td></td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00097213</td>
<td>9</td>
<td>85.67</td>
<td>0.00115910</td>
<td>9</td>
<td>98.15</td>
<td></td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00113475</td>
<td>10</td>
<td>100.00</td>
<td>0.00118089</td>
<td>10</td>
<td>100.00</td>
<td></td>
</tr>
<tr>
<td>iRPROP−</td>
<td>0.00000000</td>
<td>1</td>
<td>0.00</td>
<td>0.00000000</td>
<td>1</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.00027591</td>
<td>8</td>
<td>24.31</td>
<td>0.00049969</td>
<td>8</td>
<td>42.31</td>
<td></td>
</tr>
<tr>
<td>Batch</td>
<td>0.00018269</td>
<td>7</td>
<td>16.10</td>
<td>0.00018475</td>
<td>7</td>
<td>15.65</td>
<td></td>
</tr>
<tr>
<td>Incremental</td>
<td>0.00000000</td>
<td>2</td>
<td>0.00</td>
<td>0.00000000</td>
<td>2</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00000012</td>
<td>3</td>
<td>0.01</td>
<td>0.00000016</td>
<td>3</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.00000193</td>
<td>5</td>
<td>0.17</td>
<td>0.00000192</td>
<td>4</td>
<td>0.16</td>
<td></td>
</tr>
</tbody>
</table>

Table F.9: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the mushroom problem.
Figure F.10: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the parity8 problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train MSE</th>
<th>Rank</th>
<th>%</th>
<th>Best Test MSE</th>
<th>Rank</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 RPROP Single</td>
<td>0.00000009</td>
<td>3</td>
<td>0.00</td>
<td>0.00000009</td>
<td>3</td>
<td>0.00</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00000005</td>
<td>2</td>
<td>0.00</td>
<td>0.00000005</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.000004549</td>
<td>6</td>
<td>0.02</td>
<td>0.000004549</td>
<td>6</td>
<td>0.02</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00004549</td>
<td>6</td>
<td>0.02</td>
<td>0.00004549</td>
<td>6</td>
<td>0.02</td>
</tr>
<tr>
<td>iRPROP</td>
<td>0.000000000</td>
<td>1</td>
<td>0.00</td>
<td>0.000000000</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.00295844</td>
<td>7</td>
<td>1.18</td>
<td>0.00295844</td>
<td>7</td>
<td>1.18</td>
</tr>
<tr>
<td>Batch</td>
<td>0.250000000</td>
<td>10</td>
<td>100.00</td>
<td>0.250000000</td>
<td>10</td>
<td>100.00</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.0675762</td>
<td>9</td>
<td>27.03</td>
<td>0.0675762</td>
<td>9</td>
<td>27.03</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00000061</td>
<td>5</td>
<td>0.00</td>
<td>0.00000061</td>
<td>5</td>
<td>0.00</td>
</tr>
<tr>
<td>J neural Incr.</td>
<td>0.00347862</td>
<td>8</td>
<td>1.39</td>
<td>0.00347862</td>
<td>8</td>
<td>1.39</td>
</tr>
</tbody>
</table>

Table F.10: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the parity8 problem.
Figure F.11: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the parity13 problem.

Table F.11: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the parity13 problem.
Figure F.12: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the pumadyn-32fm problem.

Table F.12: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the pumadyn-32fm problem.
APPENDIX F. ANN BENCHMARK TEST RESULTS

Figure F.13: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the robot problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th></th>
<th>Best Test</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td>MSE</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.000000100</td>
<td>1</td>
<td>0.00</td>
<td>0.00992919</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00000107</td>
<td>2</td>
<td>0.01</td>
<td>0.01007928</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00009783</td>
<td>5</td>
<td>10.04</td>
<td>0.01056156</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00018739</td>
<td>6</td>
<td>19.33</td>
<td>0.00885875</td>
</tr>
<tr>
<td>iRPROP−</td>
<td>0.00000376</td>
<td>3</td>
<td>0.29</td>
<td>0.00856093</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.00024191</td>
<td>7</td>
<td>24.98</td>
<td>0.00983419</td>
</tr>
<tr>
<td>Batch</td>
<td>0.00096531</td>
<td>10</td>
<td>100.00</td>
<td>0.00666183</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.00001567</td>
<td>4</td>
<td>1.52</td>
<td>0.00625208</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00017563</td>
<td>8</td>
<td>28.27</td>
<td>0.00663905</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.00076423</td>
<td>9</td>
<td>79.15</td>
<td>0.0002633</td>
</tr>
</tbody>
</table>

Table F.13: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the robot problem.
Figure F.14: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the soybean problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train MSE</th>
<th>Rank</th>
<th>%</th>
<th>Best Test MSE</th>
<th>Rank</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2 RPROP Single</td>
<td>0.00025345</td>
<td>1</td>
<td>0.00</td>
<td>0.01080431</td>
<td>7</td>
<td>6.31</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00034135</td>
<td>2</td>
<td>6.71</td>
<td>0.01063425</td>
<td>6</td>
<td>5.81</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00301021</td>
<td>6</td>
<td>6.71</td>
<td>0.01377976</td>
<td>9</td>
<td>15.06</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00758661</td>
<td>9</td>
<td>17.85</td>
<td>0.01291433</td>
<td>8</td>
<td>12.52</td>
</tr>
<tr>
<td>iRPROP</td>
<td>0.04133354</td>
<td>10</td>
<td>100.00</td>
<td>0.04264027</td>
<td>10</td>
<td>100.00</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.00074942</td>
<td>4</td>
<td>1.21</td>
<td>0.01039508</td>
<td>5</td>
<td>5.10</td>
</tr>
<tr>
<td>Batch</td>
<td>0.00038203</td>
<td>3</td>
<td>0.31</td>
<td>0.00866076</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.00328510</td>
<td>7</td>
<td>7.38</td>
<td>0.01011290</td>
<td>4</td>
<td>4.27</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00331435</td>
<td>8</td>
<td>7.45</td>
<td>0.00928415</td>
<td>2</td>
<td>1.83</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.00185645</td>
<td>5</td>
<td>3.90</td>
<td>0.00963925</td>
<td>3</td>
<td>2.88</td>
</tr>
</tbody>
</table>

Table F.14: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the soybean problem.
APPENDIX F. ANN BENCHMARK TEST RESULTS

Figure F.15: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the thyroid problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th>Best Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.00002914</td>
<td>1</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00026163</td>
<td>2</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.03523034</td>
<td>9</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.03417777</td>
<td>8</td>
</tr>
<tr>
<td>iRPROP</td>
<td>0.00610040</td>
<td>6</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.02848191</td>
<td>7</td>
</tr>
<tr>
<td>Batch</td>
<td>0.04609578</td>
<td>10</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.00342221</td>
<td>4</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.00235629</td>
<td>3</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.00588516</td>
<td>5</td>
</tr>
</tbody>
</table>

Table F.15: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the thyroid problem.
Figure F.16: Graph showing an average of 4 run of the 10 different configuration discussed in section 3.3, for the two-spiral problem.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Best Train</th>
<th></th>
<th>Best Test</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSE</td>
<td>Rank</td>
<td>%</td>
<td>MSE</td>
</tr>
<tr>
<td>C2 RPROP Single</td>
<td>0.00000002</td>
<td>2</td>
<td>0.00</td>
<td>0.22243182</td>
</tr>
<tr>
<td>C2 RPROP Multi</td>
<td>0.00000006</td>
<td>3</td>
<td>0.00</td>
<td>0.22541781</td>
</tr>
<tr>
<td>C2 Quickprop Single</td>
<td>0.00000000</td>
<td>1</td>
<td>0.00</td>
<td>0.22650290</td>
</tr>
<tr>
<td>C2 Quickprop Multi</td>
<td>0.00000415</td>
<td>4</td>
<td>0.00</td>
<td>0.22818374</td>
</tr>
<tr>
<td>iRPROP⁻</td>
<td>0.13567821</td>
<td>7</td>
<td>55.87</td>
<td>0.20780282</td>
</tr>
<tr>
<td>Quickprop</td>
<td>0.10078443</td>
<td>5</td>
<td>41.50</td>
<td>0.10543431</td>
</tr>
<tr>
<td>Batch</td>
<td>0.24282845</td>
<td>10</td>
<td>100.00</td>
<td>0.24317360</td>
</tr>
<tr>
<td>Incremental</td>
<td>0.18930350</td>
<td>8</td>
<td>77.96</td>
<td>0.20895738</td>
</tr>
<tr>
<td>Lwnn Incr.</td>
<td>0.10292158</td>
<td>6</td>
<td>42.38</td>
<td>0.12539463</td>
</tr>
<tr>
<td>Jneural Incr.</td>
<td>0.24260197</td>
<td>9</td>
<td>99.91</td>
<td>0.24450797</td>
</tr>
</tbody>
</table>

Table F.16: Table showing the best reported (average of 4) train and test result, including a rank and a percent indicating how close to configurations best result is to the overall best result. The table shows the two-spiral problem.
Appendix G

Reinforcement Learning Graphs

Figure G.1: Enlargement of graph from figure 6.3 on page 101
Figure G.2: Enlargement of graph from figure 6.3 on page 101

Figure G.3: Enlargement of graph from figure 6.4 on page 102
Figure G.4: Enlargement of graph from figure 6.4 on page 102

Figure G.5: Enlargement of graph from figure 6.5 on page 103
APPENDIX G. REINFORCEMENT LEARNING GRAPHS

Figure G.6: Enlargement of graph from figure 6.5 on page 103

Figure G.7: Enlargement of graph from figure 6.6 on page 104
Figure G.8: Enlargement of graph from figure 6.6 on page 104

Figure G.9: Enlargement of graph from figure 6.7 on page 105
APPENDIX G. REINFORCEMENT LEARNING GRAPHS

Figure G.10: Enlargement of graph from figure 6.7 on page 105

Figure G.11: Enlargement of graph from figure 6.9 on page 107
Mountain Car

Figure G.12: Enlargement of graph from figure 6.9 on page 107

Mountain Car

Figure G.13: Enlargement of graph from figure 6.9 on page 107
Figure G.14: Enlargement of graph from figure 6.9 on page 107

Figure G.15: Enlargement of graph from figure 6.12 on page 112
Figure G.16: Enlargement of graph from figure 6.12 on page 112

Figure G.17: Enlargement of graph from figure 6.12 on page 112
APPENDIX G. REINFORCEMENT LEARNING GRAPHS

Figure G.18: Enlargement of graph from figure 6.12 on page 112

Figure G.19: Enlargement of graph from figure 6.14 on page 114
Figure G.20: Enlargement of graph from figure 6.14 on page 114

Figure G.21: Enlargement of graph from figure 6.19 on page 120
Figure G.22: Enlargement of graph from figure 6.19 on page 120

Figure G.23: Enlargement of graph from figure 6.19 on page 120
Figure G.24: Enlargement of graph from figure 6.19 on page 120

Figure G.25: Enlargement of graph from figure 6.21 on page 122
Figure G.26: Enlargement of graph from figure 6.21 on page 122

Figure G.27: Enlargement of graph from figure 6.23 on page 124
Figure G.28: Enlargement of graph from figure 6.23 on page 124

Figure G.29: Enlargement of graph from figure 6.24 on page 125
Figure G.30: Enlargement of graph from figure 6.24 on page 125

Figure G.31: Enlargement of graph from figure 6.25 on page 126
Figure G.32: Enlargement of graph from figure 6.25 on page 126

Figure G.33: Enlargement of graph from figure 6.25 on page 126
Figure G.34: Enlargement of graph from figure 6.25 on page 126

Figure G.35: Enlargement of graph from figure 6.25 on page 126
Figure G.36: Enlargement of graph from figure 6.26 on page 127

Figure G.37: Enlargement of graph from figure 6.26 on page 127
APPENDIX G. REINFORCEMENT LEARNING GRAPHS

Figure G.38: Enlargement of graph from figure 6.28 on page 131

Figure G.39: Enlargement of graph from figure 6.28 on page 131
Figure G.40: Enlargement of graph from figure 6.28 on page 131

Figure G.41: Enlargement of graph from figure 6.28 on page 131
Figure G.42: Enlargement of graph from figure 6.28 on page 131

Figure G.43: Enlargement of graph from figure 6.29 on page 132
Figure G.44: Enlargement of graph from figure 6.29 on page 132

Figure G.45: Enlargement of graph from figure 6.30 on page 133
Figure G.46: Enlargement of graph from figure 6.30 on page 133

Figure G.47: Enlargement of graph from figure 6.32 on page 135
Figure G.48: Enlargement of graph from figure 6.32 on page 135

Figure G.49: Enlargement of graph from figure 6.32 on page 135
Figure G.50: Enlargement of graph from figure 6.32 on page 135

Figure G.51: Enlargement of graph from figure 6.32 on page 135
Figure G.52: Enlargement of graph from figure 6.34 on page 141

Figure G.53: Enlargement of graph from figure 6.34 on page 141
APPENDIX G. REINFORCEMENT LEARNING GRAPHS

Figure G.54: Enlargement of graph from figure 6.34 on page 141

Figure G.55: Enlargement of graph from figure 6.34 on page 141
Figure G.56: Enlargement of graph from figure 6.34 on page 141

Figure G.57: Enlargement of graph from figure 6.35 on page 142
Figure G.58: Enlargement of graph from figure 6.35 on page 142

Figure G.59: Enlargement of graph from figure 6.37 on page 145
Figure G.60: Enlargement of graph from figure 6.37 on page 145

Figure G.61: Enlargement of graph from figure 6.39 on page 148
Figure G.62: Enlargement of graph from figure 6.39 on page 148

Figure G.63: Enlargement of graph from figure 6.40 on page 148
Figure G.64: Enlargement of graph from figure 6.40 on page 148

Figure G.65: Enlargement of graph from figure 6.42 on page 150
**APPENDIX G. REINFORCEMENT LEARNING GRAPHS**

Figure G.66: Enlargement of graph from figure 6.42 on page 150

**Backgammon**

- Against random player
- Against heuristic player

Figure G.67: Enlargement of graph from figure 6.42 on page 150
Figure G.68: Enlargement of graph from figure 6.42 on page 150

Figure G.69: Enlargement of graph from figure 6.42 on page 150
Figure G.70: Enlargement of graph from figure 6.43 on page 152

Figure G.71: Enlargement of graph from figure 6.43 on page 152
Figure G.72: Enlargement of graph from figure 6.43 on page 152

Figure G.73: Enlargement of graph from figure 6.43 on page 152
Bibliography


BIBLIOGRAPHY


Pashenкова, E., Rish, I., and Dechter, R. 1996. Value iteration and policy iteration algorithms for markov decision problems.


List of Acronyms

ANN  Artificial Neural Network
CMAC  Cerebellar Model Articulation Controller
DELVE  Data for Evaluating Learning in Valid Experiments
GPI  Generalized Policy Iteration
MBIE  Model-Based Interval Estimation
MBPG  Model-Based Policy Gradient
MSE  Mean Square Error
NF-SARSA  Neural Fitted SARSA Iteration
NFQ  Neural Fitted Q Iteration
NFQ-SARSA  Neural Fitted Q-SARSA Iteration
NN  Neural Network
RBF  Radial Basis Function
RBFN  Radial Basis Function Network
RL  Reinforcement Learning
SARSA  State, Action, Reward, State, Action
TD  Temporal Difference
TLU  Threshold Logical Unit
Index

$Q(\lambda)$, 66
$Q$-SARSA, 62
$Q$-SARSA algorithm, 63
$Q$-SARSA($\lambda$) algorithm, 67
$Q$-SARSA($\lambda$), 65, 66
$Q$-learning, 59
c-$\text{greedy selection}$, 52
$\lambda$-return, 64
$n$-step return, 63

Action, 1, 43
Action-selection strategy, 52
Action-value, 59
Action-value function, 47, 57, 71
Action-value reward, 57
After-state, 58, 93, 134, 173
Agent, 1, 43
ANN – Artificial Neural Network, 11, 165–171

Autonomous Land Vehicle in a Neural Net, 23
Avoidance control, 88, 92, 125
Avoidance control problems, 89
Axon, 166

Back-propagation, 11, 12
Back-Propagation algorithm, 14
Backgammon, 89
Batch algorithms, 6
Batch training, 12
Bellman optimal equation, 48
Blackjack, 89, 94
Blot, 136, 173, 174
Boltzmann-Gibbs selection, 53

Cache commit interval, 109, 158
Cache size, 109, 158
Cart pole, 87, 89, 92, 94
Cart pole problem, 2
Cascade 2, 24–27, 29
Cascade architecture, 16
Cascade-Correlation, 13, 17–19, 22–24
Cascading neural networks, 9
Catastrophic forgetting, 84
Central nervous system, 165

Cerebellar Model Articulation Controller, 11
Change fraction, 158
Classification, 9, 10, 31
Classification problems, 13, 22, 29, 30
Combined selection strategy, 57
Confidence based exploration, 70, 71
Continuous, 87
Control problem, 57, 58
Controller problem, 87, 88
Correlation, 19
Covariance, 19

Delayed Q-learning, 73
DELVE, 31–33
Dendrites, 166
Desired error, 158
Directed exploration, 54
Directed exploration methods, 61
Discount rate, 47, 64
Discounted future experience, 54, 57
Discounted future reward, 47, 57
Discrete, 87
Dyna-Q, 73
Dynamic programming, 46, 49

Eligibility traces, 7, 63–68, 79, 86, 137, 148
Ensemble learning, 12
Environment, 1, 43
Episodic, 87
Error-based exploration, 55
Experience replay, 77
Experience value, 54
Experience value function, 54, 55
Exploitation, 52
Exploitive, 52
Exploration, 52
Explorative, 52

Fast Artificial Neural Network Library, 11
Fitted Q Iteration, 80
Frequency-based exploration, 54, 56, 57
Function approximation, 9–11, 52
Step-size problem, 14–16, 21
Symmetric sigmoid activation function, 30, 40
Symmetric sinus activation function, 30, 40, 105, 106
Synaptic connection, 166

Tabu search, 70, 141
Temporal difference learning, 57
Testing data, 35
Threshold Logical Units, 13
Training algorithms, 12
  Back-Propagation, 12
  Batch back-propagation, 12
  Batch training, 12
  Bayesian techniques, 12
  Cascade-Correlation, 22
  Ensemble learning, 12
  Genetic algorithms, 12
  Incremental back-propagation, 12
  Incremental training, 12
  iRPROP−, 27, 35
  Levenberg-Marquardt, 12
  Mini-batch, 13
  Optimal brain damage, 13
  Particle swarm, 12
  Quickprop, 12, 21
  RPROP, 12, 13, 21
  Simulated Annealing, 12
Training by epoch, 12
Training by pattern, 12
Training data, 10, 35
Training patterns, 10
Trial-and-error, 43, 162
Two-spiral problem, 13

Validation data-set, 10
Value function, 47, 71
  Action-value function, 47
  State-value function, 47
Value Iteration, 49

Weight update algorithm, 17, 19, 21, 27, 29, 30, 35
Whole ANN, 84, 120, 141