Faculty of Aerospace Engineering

Guaranteed Globally Optimal Continuous Reinforcement Learning

Making self-learning controllers that always work

Hildo Bijl

June 13, 2012

TU Delft Delft University of Technology

Challenge the future

Guaranteed Globally Optimal Continuous Reinforcement Learning

Making self-learning controllers that always work

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering
at Delft University of Technology

Hildo Bijl

June 13, 2012

Faculty of Aerospace Engineering · Delft University of Technology

Delft University of Technology

Delft University Of Technology
Department Of
Control and Simulation

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"Guaranteed Globally Optimal Continuous Reinforcement Learning"** by **Hildo Bijl** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: June 13, 2012

Readers:

Prof.dr.ir. J. A. Mulder

Dr.ir. E. van Kampen

Dr. Q. P. Chu

Dr.ir. J.W. van Wingerden

Summary

Self-learning controllers offer various strong benefits over conventional controllers, the most important one being their ability to adapt to unexpected circumstances. Their application is however limited, the most important reason being that, for self-learning controllers to work on continuous domains, nonlinear function approximators are required, and as soon as nonlinear function approximators are involved, it is uncertain whether convergence will occur.

This project has as goal to contribute towards achieving convergence guarantees. The first focus lies on using reinforcement learning, combined with neural network function approximation, to create self-learning controllers. A reinforcement learning controller architecture has been set up which is capable of controlling systems with continuous states and actions. Also an extension has been made that enables the controller to freely vary its timestep without any significant consequences.

A literature research has shown that there are no convergence proofs yet of practically feasible reinforcement learning controllers with nonlinear function approximators. Several proofs of convergence for the case of linear function approximators have been provided. Also, a proof exists that a certain reinforcement learning controller algorithm with nonlinear function approximation converges. However, the corresponding learning algorithm requires an infinite amount of iterations for the value function to converge even before the policy is updated, making it practically unfeasible.

The reasons why convergence often does not occur for reinforcement learning controllers with neural network function approximators include overestimated value functions, incorrect generalization, function approximators incapable of approximating the actual value function and error amplification due to bootstrapping. Furthermore, when convergence does occur, it's not necessarily to a global optimum. Ideas to solve these issues have been offered, but it is still far from certain whether these ideas will work. Furthermore, they will make the algorithm very complex. Proving that the resulting algorithm will converge, even if it's only convergence to a local optimum, will prove to be extremely difficult, if it is at all possible. Hence reinforcement learning controllers with neural network function approximators do not seem to be appropriate if convergence to the global optimum needs to be ensured.

The literature research has also revealed that so far no attempt has been made in combining reinforcement learning with interval analysis. To investigate the possibilities of combining these two fields, the interval Q-learning algorithm has been designed. This algorithm combines

a discrete version of Q-learning with interval analysis techniques. This algorithm has proven convergence for the discrete case.

Subsequently, the discrete interval Q-learning algorithm has been expanded to the continuous domain. This was done using the main RL value assumption, which assumes that the derivatives dQ/ds_i and dQ/da_j, for every state parameter s_i and action parameter a_j, have known bounds. The resulting continuous interval Q-learning algorithm was shown to have proven convergence to the optimal value function Q^*. Furthermore, bounds on how fast the algorithm converges were given.

The most important downside of the first version of the continuous interval Q-learning algorithm was its slow run-time. This made the algorithm practically infeasible. To still meet the goals of the project, a different function approximator was designed. This function approximator used many small blocks to bound Q^* in every part of the state-action-space. Furthermore, the number of blocks was increased dynamically as the algorithm learned, thus giving the function approximator a theoretically unlimited accuracy. Though the resulting algorithm used information slightly less efficiently than its precursor, its run-time was significantly improved. It became practically feasible to apply this algorithm.

In the end of the report, the algorithm has been applied to a few simple test problems. An important parameter here was the dimension D of the problem, which equals the sum of the number of state and action parameters. For two- and three-dimensional problems, the algorithm was able to sufficiently bound the optimal value function Q^* quite quickly, resulting in a controller with satisfactory performance. For the cart and pendulum system, which is a five-dimensional problem, this turned out to be different. A long training (in the order of several hours or more) will be required before a satisfactory performance can be obtained. However, since the algorithm has proven convergence to the globally optimal policy, this does not necessarily have to be a big problem.

Finally, it is mentioned that this thesis report has introduced the world's first combination of reinforcement learning and interval analysis. It also introduced the world's first practically feasible continuous RL controller with proven convergence to the global optimum. The key to accomplishing such a controller turned out to be (A) letting go of conventional ways of designing continuous RL controllers, and (B) quantifying the assumption that the value Q of two nearby states s and s' are similar through the main RL value assumption.

Acronyms

ANN	Artificial neural network
CAP	Cart and pendulum
CF	Centered form
CMAC	Cerebella model articulation controller
DHP	Dual heuristic programming
GDHP	Global dual heuristic programming
GMM	Gaussian mixture model
HDP	Heuristic dynamic programming
IA	Interval analysis
LQR	Linear quadratic regulator
LSPI	Least-squares policy iteration
MAS	Multi-agent system
MDP	Markov decision process
MSE	Mean-squared error
MTF	Monotonicity test form
MVF	Mean value form
NDP	Neuro-dynamic programming
NEAT	Neuro-evolution of augmenting topologies
NN	Neural network
PDF	Probability density function
RBF	Radial basis function
RGD	Restricted gradient-descent
RL	Reinforcement learning
RMS	Root mean square
RRL	Relational reinforcement learning
SF	Slope form
TD	Temporal difference

Contents

Part I

Introductory matter

Chapter 1

Introduction

In the world of control theory, controllers are usually static programs. They cannot adjust to unexpected circumstances. Self-learning controllers, due to their learning abilities, can cope with unexpected events. Applying self-learning controllers thus seems like a significant step forward, not only towards making control systems more effective, but also to making them more safe. Possible applications, for example in the world of aerospace engineering, are legion.

The reason why self-learning controllers, and specifically reinforcement learning controllers with function approximators, aren't broadly applied yet, is because they have several issues. The current project has as aim to contribute towards solving these issues. First the issues themselves are investigated. Then a literature survey is conducted on how far along mankind is at solving them. Finally an additional contribution is made towards solving the issues involved in continuous reinforcement learning.

Part I offers an introduction into the problem to be solved. This is mainly done in chapter 2, which examines the problem to be solved, as well as what needs to be accomplished to solve the problem. Next, part II offers summarized background knowledge on all required techniques used in the remainder of this report. Chapter 3 is on reinforcement learning, chapter 4 discusses neural networks, chapter 5 discusses combining reinforcement learning with function approximation and chapter 6 contains a summarized introduction into interval analysis. The readers that already have background knowledge in these fields may skip this part of the report.

Part III discusses a literature research on reinforcement learning with function approximation. Chapter 7 contains a rather specific literature research focused on the exact problem to be solved. At the end of this chapter is a summary on the findings of this literature research, listing which problems arose in literature and what the possible solutions are. On the other hand, chapter 8 goes into depth on subjects less directly related to the problems involved with reinforcement learning and function approximation. However, as can be read in this chapter, fields not directly related to reinforcement learning can still provide ample inspiration to improve the reinforcement learning algorithm.

Next, part IV offers an initial problem analysis. Chapter 9 offers a discussion on setting up a reinforcement learning controller for a cart and pendulum system. It looks at the issues

involved in doing so and then examines whether it is possible to prove (global) convergence of such an algorithm. Chapter 10 offers a completely different approach. It examines whether it is possible to combine interval analysis and Q-learning. The subsequent interval Q-learning algorithm for discrete RL systems shows that such a combination is indeed possible.

In the final part, part V, the discrete interval Q-learning algorithm is expanded to the continuous domain, resulting in the continuous interval Q-learning algorithm. Chapter 11 sets up the theory behind the algorithm and also proves that it indeed always converges to the global optimum. Chapter 12 then shows that the designed algorithm is too slow to be practically feasible. It then continuous by developing an alternative function approximator, which is only slightly less efficient in its data usage, but is fast enough to be practically feasible. The resulting algorithm is then applied to several practical problems. In the end, in chapter 13, some conclusions and recommendations are given.

Finally, there are also two appendices. Appendix A contains various mathematical theorems and proofs, while appendix B discusses the author's thoughts on developing human-like artificial intelligence.

It is hoped that this thesis report contributes towards improving reinforcement learning controllers and brings them one step closer towards actually being implemented in the world around us.

Chapter 2

Problem statement

Self-learning controllers are a very promising type of controllers. One of the reasons they are not yet so widespread is because there are still some issues in their implementation. In this chapter, these issues will be examined, as well as what needs to be done to solve them. Section 2-1 will look at some general complications involved in controlling systems using self-learning controllers. After that, section 2-2 discusses the working principle of a basic cart and pendulum system, while section 2-3 discusses the goals of the current project and of this report.

2-1 Issues in self-learning controllers

Imagine that self-learning controllers would work well, and would be widely applied across this planet. What benefits would this have? Of course it would prevent the need to make adequate controllers for every system all the time. It would save a lot of time, manpower and money. But that is not the only (and not even the most important) benefit.

In aviation, it regularly occurs that an airplane encounters some unexpected phenomenon. For example, an engine may fall off as occurred during the Bijlmercrash (El Al Flight 1862) in October 1992, see (Raad voor de Luchtvaart (Netherlands Aviation Safety Board), 1994), the airplane might fly too slowly as was the case for the Buffalo crash (Colgan Air Flight 3407) in February 2009, see (Aviation Safety Network, 2009), or something as simple as a pitot tube icing may occur, as happened with Air France Flight 447, see (BEA (Bureau d'Enquêtes et d'Analyses pour la sécuritéde l'aviation civile), 2011). If something like this happens, the autopilot realizes it has not been designed for the current situation and shuts off. The pilots are thrust back in control. However, often the pilots know less about the current situation than the autopilot does. Subsequently, it often happens that the pilots choose the wrong course of action and crash the airplane, as has happened in the three previously mentioned examples.

It would be much better if the autopilot would be able to adapt to unknown circumstances. Conventional controllers cannot do such a thing. Self-learning controllers can. They can learn

how to cope with new unknown circumstances on the fly. Or what would be even better: they could be trained to handle thousands, if not millions, of different possible unexpected circumstances. Giving every pilot in the world such a training would be economically (if not practically) impossible. Giving one computer such a training and then copying the result to all airplanes would be possible. If an unexpected situation then arises, the smart autopilot would recognize it from its earlier training and quickly adapt in the optimal way – something a pilot could rarely, if ever, do.

So why aren't these self-learning controllers applied? The reason is that they are not reliable enough yet. When these self-learning controllers are trained, they sometimes learn the right behavior, but sometimes they do not. It can occur that the learning algorithm converges to a sub-optimal behavior, to an erroneous behavior, or it may not converge at all. This phenomenon has occurred countless of times in literature, but it still isn't well-understood, nor (as will be seen in chapter 7 of this report) has anyone found a conclusive solution yet.

If a self-learning controller would ever be applied on an aircraft, there need to be guarantees that it learns the right behavior. Without such guarantees, it would be impossible to get the controller certified. One goal of the current project is thus to explore ways with which convergence of self-learning controllers to the optimal behavior might be ensured. This report provides a preliminary exploration on the subject.

2-2 The cart and pendulum system

Once a controller has been made, it needs to be tested on a system. One very popular and well-known system in the world of control theory is the **cart and pendulum** (CAP) system. According to (Russell & Norvig, 2003), *'over two thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem.'* This section briefly discusses how the system works and will continue on why this is such a relevant problem.

2-2-1 The dynamics of the cart and pendulum system

The cart and pendulum system consists, as might have been guessed, of a cart with a pendulum on top of it, as shown in Figure 2-1. The goal of the system is first of all to keep the pendulum up-right. In other words, the clockwise rotation α should be kept at 0 degrees. If the angle becomes too big (say, $|\alpha| > 10°$), the simulation is terminated. The second goal of the system is to keep the horizontal position of the cart at $x_c = 0$. If the cart is too far off (say, $|x_c| > 1$ m), the simulation is terminated as well.

The CAP system has various parameters. First of all, there is the length L of the (massless) pendulum. In this report, it has been set to $L = 1$ m. There are also the mass of the cart $m_c = 4$ kg and the mass of the ball attached to the pendulum $m_b = 1$ kg. Finally, there is the gravitational coefficient $g = 9.81$ m/s^2.

There is only one way to control the cart and the pendulum: through a horizontal force F applied to the cart. This force is bounded by $|F| \leq F_{max} = 10$ N and should be used to satisfy the two goals mentioned earlier. Before it can be figured out how this can be done,

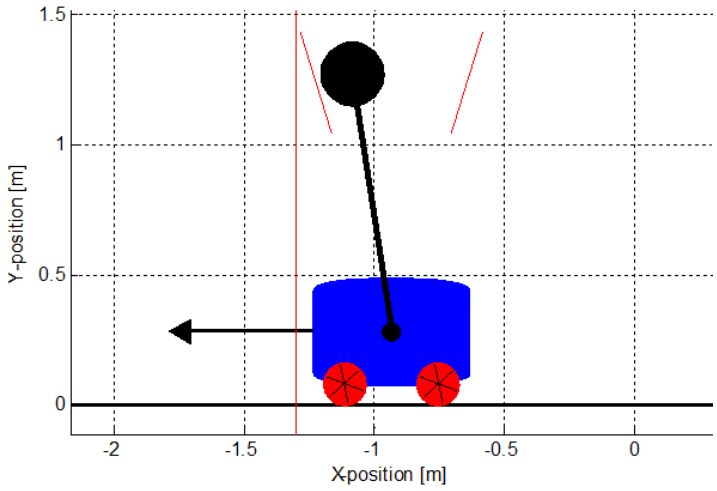

Figure 2-1: Visualization of the cart and pendulum system. Visible are the cart, the pendulum, the controlling force F and the terminating boundaries.

first the dynamics of the system need to be analyzed. It can be shown using basic mechanics that $\ddot{\alpha}$ equals

$$\ddot{\alpha} = -\frac{\dot{\alpha}^2 L \sin(\alpha)\cos(\alpha) m_b + F\cos(\alpha) - (m_b + m_c) g \sin(\alpha)}{L(m_c + m_b \sin^2(\alpha))}. \tag{2-2-1}$$

Once $\ddot{\alpha}$ has been found, also the horizontal force F_x that is exerted between the pendulum and the cart can be found. It equals

$$F_x = \frac{m_b}{m_b + m_c}\left(m_c L \cos(\alpha)\ddot{\alpha} + F - m_c L \sin(\alpha)\dot{\alpha}^2\right). \tag{2-2-2}$$

With this force, it is easy to find the second derivative of the horizontal cart position x_c. It is given by

$$\ddot{x}_c = \frac{F - F_x}{m_c}. \tag{2-2-3}$$

By numerically integrating $\ddot{\alpha}$ and $\ddot{x}_c$, the four state parameters α, $\dot{\alpha}$, x_c and $\dot{x}_c$ can be found. Since simulation accuracy is not a necessity here, the numerical integration simply takes place by multiplying the derivatives by the relatively small time step.

2-2-2 Goals of the controller

One important goal of this report is to write a controller that adequately learns to control that CAP system. With 'adequate', several things are meant. The controller should...

- Keep the pendulum up-right.
- Get the cart near the reference position. Even in case of a changing reference position, the cart should follow the reference position.
- Do the above without any significant (unnecessary) oscillations.

- Do the above without any significantly high input values.

Furthermore, the controller will not be given any data on the dynamics of the system. It should figure this out entirely on its own. Of course the controller will know the dimension of the state vector, as well as the dimension of the input vector, but no system dynamics are provided.

So how can the controller learn the system behavior? It will simply have to provide an input F to the system, observe what happens to the state s and learn from this. From this, it should then learn how to control the system. If possible, it should to this in a reasonable time. What is more important though, is that convergence is guaranteed. The chance that the wrong behavior is learnt should be minimized, or preferably even avoided altogether.

2-2-3 The relevance of the cart and pendulum system

The question remains: why is the CAP system so popular? One of the most important reasons is that it is a very simple system that still contains many of the same characteristics as its more complicated counterparts.

First of all, there is the integrative (delay) kind of behavior. The input force F only affects $\dot{x}_c$ and $\dot{\alpha}$. The parameters to be controlled, however, are x_c and α. This is similar to an aircraft where, for example, the elevator deflection only affects the pitch rate q, while the actual parameter to be controlled is the pitch angle θ or the angle of attack α.

Secondly, the CAP system has a complicated dynamic behavior. To see how, imagine that the pendulum is positioned perfectly upright ($\alpha = 0$) at $x_c = 0$ m and needs to be moved to $x_c = 1$ m. To achieve this, the cart needs to be moved to the right, but if this is done directly, the pendulum will fall to the left. Instead, the cart should first be moved to the left (in the opposite direction) to cause the pendulum to fall to the right. Then, to 'catch' the pendulum, the cart can also be moved to the right. Subsequentially, the cart should overshoot the desired position $x_c = 1$ m a bit, slow down and move back to $x_c = 1$ m to once more get the pendulum upright.

This behavior of 'first going into the wrong direction' is a type of behavior that not all controllers can learn. However, such behavior is present in an airplane. (For example, when you're in a stall and want to gain altitude, you should first dive down; i.e., move in the 'wrong' direction.) So any controller architecture that might one day aspire to control an airplane should easily be able to cope with the relatively simple CAP system.

So it is seen that the CAP system is a simple system that is easy to analyze, but still exhibits many of the same complicated types of behaviors as an actual airplane. It is thus ideal to test potential controllers on. One more goal of this report is to find out how to set up and then actually produce a self-learning controller that is able to learn how to control a CAP system, subject to the requirements mentioned in the previous subsection. Subsequentially, some ideas should be postulated on how it might be possible to ensure that the self-learning controller always learns the optimal behavior.

2-3 Goals of the project and of this report

The current project has various goals. Some have already been described earlier in this chapter. Now all goals are stated below.

- Examine how self-learning controllers can be applied to continuous systems like aircraft. Also explore any arising issues.
- Find ways to combine interval analysis with reinforcement learning. Interval analysis is a promising technique that is often used to bound parameters. It might be used to ensure that the reinforcement learning algorithm stays bounded and actually converges. How this should be done is something that needs to be examined.
- Explore ways with which convergence of self-learning controllers to the optimal behavior might be ensured.
- Set up a self-learning algorithm and prove its convergence, either to a local optimum or (preferably) to the global optimum.
- Test the resulting algorithm on the representative cart and pendulum system to analyze its performance.

This report describes the process of achieving these goals throughout the research project.

Part II

Background knowledge

Chapter 3

Reinforcement learning

Reinforcement Learning (RL), in some literature also called **Neuro-Dynamic Programming** (NDP), is a technique with which learning controllers can be made. This chapter introduces the technique and discusses its possibilities and limitations. Concepts in this chapter have mainly originated from (Babuška, 2010), (Sutton & Barto, 1998) and own derivations, though (Russell & Norvig, 2003), (Qiang & Zhongly, 2011) and (Webb, 2002) have also contributed insights.

3-1 Basics of reinforcement learning

3-1-1 Definitions in reinforcement learning

In reinforcement learning (RL), there is an **agent** and an **environment**. At every time instance k, the agent has a certain **state** $s_k \in S$. During every step, the agent needs to choose one of the possible **actions** $a_k \in A$. It then reaches a new state s_{k+1}. By doing this, it gets an **immediate reward** $r_{k+1} \in \mathbb{R}$ from the environment.

The goal of the agent is to maximize the **total reward** R_k. This total reward is a function of all future rewards. Often, the sum is used. So, $R_k = r_{k+1} + r_{k+2} + \ldots$. Another often-used function is

$$R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \ldots = \sum_{n=0}^{\infty} \gamma^n r_{k+n+1}. \tag{3-1-1}$$

The parameter γ, which satisfies $0 \leq \gamma \leq 1$, is called the **discount rate**, or sometimes also the **decay factor**. It basically says that rewards obtained in the near future are more valuable than rewards obtained in the far future. Note that, if $\gamma = 1$, then again the first definition of the total reward is used. In this report, only the discounted version of the total reward is considered, since it is more general.

The main idea behind reinforcement learning is to find the optimal **policy**. A policy is a mapping: for every state s, it maps which action a is chosen by the agent in that state. For

some problems it is possible to write $a = \pi(s)$. That is, the action a is uniquely defined by the state s. Such a policy is called a **deterministic policy**: for every state s, always the same action a is chosen. However, for some problems it's better to use a **stochastic policy**. In this case, the policy is described by $\Pi(s, a)$, where $\Pi(s, a)$ denotes the probability that in state s action a is chosen by the agent.

3-1-2 The environment

The amount of knowledge on the environment that is available varies per problem. In some problems, the agent doesn't fully know the state of the environment. (Think of games like poker.) Such problems are called **partially observable problems**. These types of problems are very difficult to deal with and will not be discussed further in this report. Instead, this report only considers **fully observable problems**, in which the state of the environment is always fully known. (An example of a fully observable problem is chess)

Now consider an agent that is in some state s_k and chooses an action a_k. Also, all the previous states and actions $s_{k-1}, a_{k-1}, s_{k-2}, a_{k-2}, \ldots$ are known. In a **deterministic environment**, it is always certain which state s_{k+1} the agent winds up in. (Again, think of chess.) This is not the case in problems (like for example backgammon) with a **stochastic environment**. Now, the probability that the agent reaches the state s_{k+1} is denoted by

$$P\left(s_{k+1}|s_k, a_k, s_{k-1}, a_{k-1}, s_{k-2}, a_{k-2}, \ldots\right). \tag{3-1-2}$$

Consider the above relation. As times progresses, the list of past state and actions $s_k, a_k, \ldots$ becomes very long and cumbersome. Therefore, it is often assumed that the system has the **Markov property**. This means that the new state and reward at time $k + 1$ only depend on the state and action at time k. Thus, the above probability is simply written as $P\left(s_{k+1}, r_{k+1}|s_k, a_k\right)$. An RL task which satisfies this property is called a **Markov decision process** (MDP). In this report only Markov decision processes will be considered.

For MDPs, a shorter notation can be used. The **state transition probability function** is written as

$$\mathcal{P}^a_{ss'} = P\left(s_{k+1} = s'|s_k = s, a_k = a\right). \tag{3-1-3}$$

Similarly, the **expected reward** is defined as

$$\mathcal{R}^a_{ss'} = E\left\{r_{k+1}|s_k = s, a_k = a, s_{k+1} = s'\right\}. \tag{3-1-4}$$

3-1-3 The value function

Again, consider an agent in some state s. This agent also has a policy π. The **value function** $V^\pi(s)$ is defined as the expected total reward R_k when the policy π is used. So,

$$\begin{aligned} V^\pi(s) &= E^\pi\left\{R_k|s_k = s\right\} \\ &= E^\pi\left\{\sum_{n=0}^{\infty} \gamma^n r_{k+n+1}|s_k = s\right\}, \end{aligned} \tag{3-1-5}$$

where E^π is the expectation operator, given that the agent follows the policy π. (Note that for deterministic problems the expectation operator can be omitted from the above equation.)

In a similar way, the **action-value function** $Q^\pi(s,a)$ is defined as the expected total reward R_k when an agent chooses action a in state s and follows the policy π afterwards. So,

$$\begin{aligned} Q^\pi(s,a) &= E^\pi\left\{R_k|s_k=s,a_k=a\right\} \\ &= E^\pi\left\{\sum_{n=0}^{\infty}\gamma^n r_{k+n+1}|s_k=s,a_k=a\right\}. \end{aligned} \tag{3-1-6}$$

When applying RL, always either V or Q is used, but never both. As will be seen later, sometimes V is more convenient to use, while sometimes Q works better. Since this varies per problem, for now both V and Q will be discussed.

The goal of reinforcement learning is to find an **optimal policy** π^*. This optimal policy π^* is the policy π which maximizes the value function V^π or, alternatively, Q^π. How this policy can be found depends on the type of problem. Several types of problems will be considered in the upcoming sections.

3-2 Model-based RL

In **model-based RL** it is assumed that a model of the environment is known. That is, the **state transition function** $P\left(s_{k+1}|s_k,a_k\right)$ is known, or accurately approximated. The whole point of an RL agent is that it learns to choose good actions. How can it do that? That will be discussed now.

3-2-1 The Bellman optimality equation

Imagine that the agent is in a state s and chooses an action a. If it does this, then there is a chance $\mathcal{P}^a_{ss'}$ that it winds up in the state s'. In this state, the expected total reward $E\left\{R_k\right\}$ will be the sum of the immediate reward $\mathcal{R}^a_{ss'}$, and of the expected total reward of future states $\gamma V^*(s')$. (Note that a discount rate has to be added.) Of course, the agent wants to choose the action a which maximizes its expected total reward. This logic results in the recursively defined **Bellman optimality equation**

$$V^*(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma V^*(s')\right). \tag{3-2-1}$$

A similar equation can be derived for Q. This gives

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}\left(\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')\right). \tag{3-2-2}$$

The trick is to find the value function V or Q that satisfies this optimality equation. This can be very hard, as will be seen in the remainder of this section.

Suppose for now that the optimal value function V^* or Q^* is found. How can one then find the optimal policy? In this case the optimal policy is the so-called **greedy policy**. The agent

simply takes the action a that maximizes the value function (i.e. the expected total reward). In an equation this becomes,

$$\pi(s) = \arg\max_{a \in A} \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V^\pi(s) \right), \tag{3-2-3}$$

$$\pi(s) = \arg\max_{a \in A} Q^*(s, a). \tag{3-2-4}$$

3-2-2 Finding the optimal value function

There are two often-used methods to find the optimal value function V^* or Q^*: policy iteration and value iteration.

In **policy iteration**, one starts off with a certain initialization $V_0(s)$ of the value function and with a certain policy π. Then several iterations are performed, each containing a **policy evaluation** step and a **policy improvement** step. In the policy evaluation step, the current policy π is used to update the value function. This is done according to

$$V_{n+1}(s) = E^\pi \left\{ r_{k+1} + \gamma V_n(s_{k+1}) \right\} = \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V_n(s') \right), \tag{3-2-5}$$

with $a = \pi(s)$. This is continuously done for all states s until the value function $V_{n+1}(s)$ converges. Next, in the policy improvement step, the new value function is used to improve the policy π. In fact, the new policy will be the greedy policy π, corresponding to the new value function $V_{n+1}(s)$. It can be shown that every policy results in improved performance. Eventually the algorithm is guaranteed to converge to the optimal policy π^*.

The policy iteration method has a drawback. Sometimes, the policy evaluation step takes a long time to converge. A solution is the **value iteration** method. In this method, one does not wait for the value function to converge. Instead, one simply updates the value function $V_{n+1}(s)$ once for every state s, after which the policy improvement step immediately commences. This simplification effectively turns the whole iteration into one equation, being

$$\begin{aligned} V_{n+1}(s) &= \max_a E \left\{ r_{k+1} + \gamma V_n(s_{k+1}) | s_k = s, a_k = a \right\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma V_n(s') \right). \end{aligned} \tag{3-2-6}$$

3-3 Model-free RL

Previously, it was assumed that a model of the environment was known. This is not always the case though. It is also possible to drop an agent into an unknown world. The only way in which it can then learn is by exploring: trying actions and seeing what rewards it will get. But in what way should it explore, and how does it learn from it? In this section several solutions to those questions are discussed.

3-3-1 Temporal difference methods

The most often used methods in model-free RL are the **temporal difference** (TD) methods. Suppose that the agent is in some state s_k. It then goes to a state s_{k+1} in which it receives a reward r_{k+1}. In TD methods, this reward is used to update $V(s_k)$. This actually makes sense: if r_{k+1} is big, then $V(s_k)$ should also have been big, while if r_{k+1} is small, then $V(s_k)$ should have been small as well.

The equation that is used in TD methods is

$$\begin{aligned} V(s_k) &\leftarrow (1-\alpha_k)V(s_k) + \alpha_k\left(r_{k+1} + \gamma V(s_{k+1})\right) \\ &= V(s_k) + \alpha_k\left(r_{k+1} + \gamma V(s_{k+1}) - V(s_k)\right) \\ &= V(s_k) + \alpha_k \delta_k. \end{aligned} \tag{3-3-1}$$

In the above equation, α_k is the **learning rate** at time k. Also, $\delta_k = r_{k+1} + \gamma V(s_{k+1}) - V(s_k)$ is a quantity known as the **TD-error**.

3-3-2 Eligibility traces

One might be wondering, why is r_{k+1} used to only update $V(s_k)$. Isn't it better to use r_{k+1} to update $V(s_{k-1}), V(s_{k-2}), \ldots$ as well? Well, it is. The question just is: how much should they be updated?

For this, the **eligibility trace** $e_k(s)$ is used. This eligibility trace can be seen as the 'strength' of the relation between the reward r_{k+1} and the state s. If the state s occurred very recently (for example, $s = s_{k-1}$) then there is a relatively strong relation between s and r_{k+1}, so $e_k(s)$ should be big. On the other hand, if (for example) $s = s_{k-20}$, then $e_k(s)$ should be small. So $e_k(s)$ can be defined as

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1}(s) & \text{if } s \neq s_k, \\ 1 & \text{if } s = s_k. \end{cases} \tag{3-3-2}$$

The parameter λ is called the **trace-decay parameter**. (γ is still the discount rate.) Based on this eligibility trace, the value function $V(s)$ can be updated. The change in $V(s)$ (denoted as $\Delta V(s)$) is now given by

$$\Delta V(s) = \alpha \delta_k e_k(s). \tag{3-3-3}$$

The above method is called **replacing eligibility traces**: every time a state s is visited, its trace is replaced by the number 1. Alternatively, one can also use **accumulating eligibility traces**. This turns the update relation into

$$e_k(s) = \begin{cases} \gamma\lambda e_{k-1}(s) & \text{if } s \neq s_k, \\ \gamma\lambda e_{k-1}(s) + 1 & \text{if } s = s_k. \end{cases} \tag{3-3-4}$$

In this way, when a state s is visited more often, eligibility traces accumulate for that state. Depending on the problem to be solved, this might make the RL agent learn faster or less fast.

3-3-3 Q-learning

Another model-free RL method is **Q-learning**. It is a so-called **off-policy** method: it learns the Q-function for the greedy policy, even when it uses a totally different policy. To start, the function $Q(s,a)$ is given initial values. It is then updated using

$$\begin{aligned} Q(s_k,a_k) &\leftarrow Q(s_k,a_k) + \alpha\left(r_{k+1} + \gamma \max_a Q(s_{k+1},a) - Q(s_k,a_k)\right) \\ &= Q(s_k,a_k) + \alpha\delta_k. \end{aligned} \tag{3-3-5}$$

How does this work? Suppose that the value $Q(s_k,a_k)$ requires updating. The agent then starts in the state s_k and chooses action a_k, bringing it to the state s_{k+1} with immediate reward r_{k+1}. This data is used to update $Q(s_k,a_k)$. The new value depends on the old value, the immediate reward r_{k+1} and the maximum expected future reward $Q(s_{k+1},a)$ which the agent expects to be able to get. This method has proven convergence, as long as all state-action pairs (s_k,a_k) are continually visited during execution of the learning algorithm.

It is also possible to add eligibility traces to the Q-learning method. This turns the above equation into

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_k e_k(s,a). \tag{3-3-6}$$

3-3-4 SARSA

Another method, which is somewhat similar to Q-learning, is the **SARSA** method. (SARSA stands for State, Action, Reward, State, Action.) But contrary to Q-learning, SARSA is an **on-policy** method. That is, it does follow a policy π or Π, and it learns the Q-function for that particular policy.

How does it work? The agent simply walks around in its environment according to its policy π (or Π). So when it is in state s_k, it select the action $a_k = \pi(s_k)$. (Or similarly, the chance that it chooses a_k equals $\Pi(s_k)$.) It then updates the value function $Q(s_k,a_k)$ according to

$$Q(s_k,a_k) \leftarrow Q(s_k,a_k) + \alpha\left(r_{k+1} + \gamma Q(s_{k+1},a_{k+1}) - Q(s_k,a_k)\right). \tag{3-3-7}$$

The Q-learning method is very suitable when it is easy to change the state s_k to any required state. (For example, it is easily possible to place the robot agent at any position in the maze.) If this is not possible, and the only way to change the state is by choosing an action, then the SARSA method is more suitable.

3-3-5 Exploration

When applying the SARSA method, which policy π should be used? That is a difficult question. It is important that all states s are visited. One option is to always choose **greedy actions** which have the highest expected rewar, but this is likely to prevent the agent from reaching certain states s or trying certain actions a. It could be missing out on a very good action a.

Instead, to learn well, the agent should also do **explorative actions**. These actions have a lower expected reward, but the goal of choosing them is to learn. It might just be possible

to find an action that has a very high reward, which will improve the rewards of the greedy actions. The trick now is to find a good balance between greedy and explorative actions. There are several ways to do this. The most well-known type of methods are known as **undirected exploration** methods. The word 'undirected' simply means that chances are used in selecting actions.

The most simple undirected exploration method is the **ϵ-greedy policy**. In this method, there is a chance ϵ that a random action is selected. In the other case (with chance $1-\epsilon$), a normal greedy policy is followed, and thus the action with the highest $Q(s,a)$ value is chosen.

Another type of undirected exploration is **Max-Boltzmann exploration**, also called **soft-max exploration**. Now the chance that an action a is chosen is given by

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}}. \tag{3-3-8}$$

The parameter τ is a variable that determines how much is explored. If $\tau = \infty$, actions are selected fully randomly (as if $\epsilon = 1$), but if $\tau = 0$, a greedy policy is used.

One could also use **optimistic initial values**. What this means is that $Q(s,a)$ (or alternatively, $V(s)$) is given very high initial values. Subsequently, a greedy policy with a normal updating method for Q is used. When the agent tries an action a, the $Q(s,a)$ value will very likely decrease, so the next time it arrives at state s, it will choose a different action. Only if, for some action a, the value of $Q(s,a)$ stops to decrease, will the agent continue to choose the same action a. This is then quite likely the best action.

A very interesting question to ask is: how much should be explored? This is called the **exploration vs. exploitation dilemma**. Initially the agent should explore quite a bit. But, as time progresses, and the agent probably has found some good sets of actions, it should also exploit its knowledge, to cash in on the high rewards. Thus, when applying an ϵ-greedy policy or a Max-Boltzmann policy, the value of ϵ or τ should decrease over time.

Chapter 4

Neural networks

Reinforcement learning is a promising technique, but it has a downside. When the number of states increases, or when the state is continuous (and there are thus infinitely many possible states) it is computationally impossible to calculate the value for every state. The solution to this problem is to use a function approximator.

One such a function approximator is an **(Artificial) Neural Network** (ANN/NN). ANNs are imitations of biological neural networks, like human brains. They can be very adept at approximating nonlinear functions, even when only few training data is available. This chapter introduces the basic principles of ANNs, starting with individual neurons, building up to complete neural networks, continuing with the backpropagation learning algorithm and ending with possible improvements to NNs. Concepts in this chapter have mainly originated from (Babuška, 2010), (Haykin, 1999), (Zalzala & Morris, 1996), (Graupe, 1997), (Russell & Norvig, 2003) and own derivations.

4-1 The basic set-up of a Neural Network

4-1-1 The neuron

The basic building block of an ANN is a **neuron**, as shown in Figure 4-1. A neuron has several inputs x_i. Each of these inputs is multiplied by a weight w_i and then added up. Often, a bias b is added as well. The result is the neuron's **activation** z. So,

$$z = \sum_{i=1}^{p} w_i x_i = \mathbf{w}^T \mathbf{x} \qquad \text{or} \qquad z = \sum_{i=1}^{p} w_i x_i + b = \begin{bmatrix} \mathbf{w}^T & b \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}. \tag{4-1-1}$$

From the above equation, it can be seen that adding a bias b works the same as adding an additional input with weight b and value 1. Therefore, the relation $z = \mathbf{w}^T \mathbf{x}$ can also incorporate the bias and will thus be used in the remainder of this report.

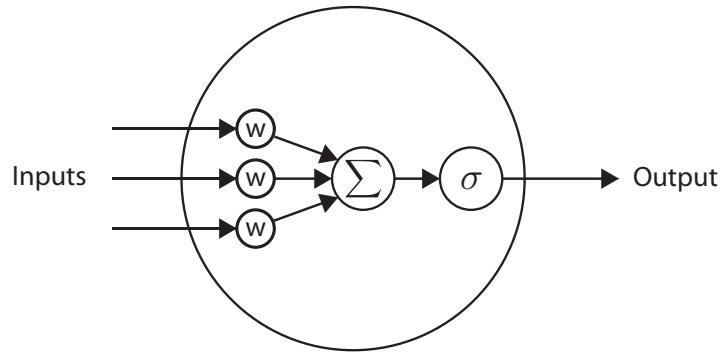

Figure 4-1: Overview of a Neuron. Inputs are multiplied by weights, summed up and fed into a sigmoidal function. The result is the Neuron output.

Once the neuron's activation z has been obtained, it is fed into the **activation function** $\sigma(z)$. Which activation function is used depends on the ANN designer's choice. Common activation functions are the **threshold function** and the **sigmoidal function**, respectively defined as

$$\sigma(z) = \begin{cases} -1 & \text{for } z < 0 \\ 1 & \text{for } z \geq 0 \end{cases} \qquad \text{and} \qquad \sigma(z) = \frac{1 - \exp(-sz)}{1 + \exp(-sz)}, \tag{4-1-2}$$

or the **Radial Basis Function** (RBF) $\phi(||\mathbf{x} - \mathbf{c}||)$, where often

$$\phi(r) = \exp\left(-(\sigma r)^2\right). \tag{4-1-3}$$

Also, c is the RBF center, while σ determines the width of the RBF. In this report, the sigmoidal function as defined above will be used. In this function, the parameter s determines how 'steep' the sigmoidal function is. If $s \to \infty$, then the threshold function is again obtained. In this report $s = 1$ is used. The output of the activation function $\sigma(z)$ subsequently becomes the output y of the neuron.

4-1-2 The neural network architecture

An artificial neural network consists of interconnected neurons. (That is, the output of one neuron is fed as input to the next neuron.) The neurons are usually assembled in layers. In a **feedforward network**, the neurons of every layer are connected to the next layer. On the other hand, in **recurrent networks**, neurons are also connected to previous layers as some sort of 'feedback mechanism'. When using an ANN as a function approximator in an RL scheme, the advantages of feedback mechanisms are very limited, so therefore this report will only consider feedforward networks. An example of a feedforward network is shown in Figure 4-2.

NNs always have an **input layer** (at the start) and an **output layer** (at the end). There are also **hidden layers** in between. In this report, only one hidden layer is used. The reason is that multiple hidden layers make the neural network computationally much more complex, while one (sufficiently large) hidden layer is already capable of approximating any continuous function. It has also been verified with tests (which for reasons of brevity will not be elaborated here) that using multiple hidden layers will significantly increase the runtime,

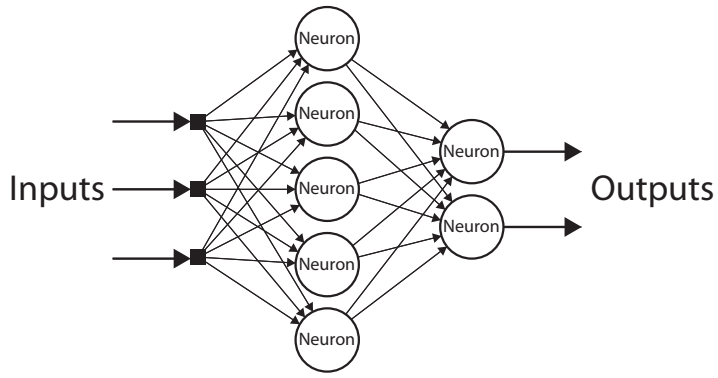

Figure 4-2: Overview of a neural network. Inputs are fed into the input layer, pass through one or more hidden layers and are then fed through the output layer as output of the neural network.

while not having a significant improvement on performance. Nevertheless, for generality, this chapter will still explain the theory of NNs with multiple hidden layers.

Every hidden layer has a number of neurons. Choosing this number is very important, but also very difficult. If one uses too few neurons, then the neural network can't approximate the desired output function well enough. If however one uses too many neurons, then overtraining may occur: the system only works on the few test samples that have been provided, but is useless for any other input. Furthermore, it would increase the runtime of the learning algorithm. Generally, the number of hidden neurons primarily depends on the number of training samples (having more training samples implies that more neurons can be used) and the complexity of the output function (more complex output functions often require more neurons). Designer's experience and intuition play a large role in deciding this number.

4-1-3 Finding the output of a neural network

Consider again the network shown in Figure 4-2. Given an input $\mathbf{x}$, how is the output $\mathbf{y}$ of this network defined? To answer this question, one starts at the input layer. This input layer doesn't really do anything with this input. It only passes it on to every neuron of the first hidden layer.

Next, consider any hidden layer j. The input to hidden layer j is denoted by $\mathbf{x_j^h}$. (Note that $\mathbf{x_1^h} = \mathbf{x}$.) The **weight matrix** of this layer is given by

$$W_j^h = \begin{bmatrix} \mathbf{w_{j1}^h}^T \\ \mathbf{w_{j2}^h}^T \\ \vdots \\ \mathbf{w_{jp_j}^h}^T \end{bmatrix}, \tag{4-1-4}$$

where p_j is the number of neurons in layer j. Also, $\mathbf{w_{ji}^h}$ is the weight vector of neuron i in hidden layer j of the network. The activation z_{ji} of any hidden neuron i in this layer j can now be found using $z_{ji} = \mathbf{w_{ji}^h}^T \mathbf{x_j^h}$. But of course there are multiple hidden neurons. Therefore, the general equation for the **layer activation** $\mathbf{z_j}$ of layer j is

$$\mathbf{z_j} = W_j^h \mathbf{x_j^h}. \tag{4-1-5}$$

The **layer output** $\mathbf{y_j}$ of layer j is then given by $\mathbf{y_j^h} = \sigma(\mathbf{z_j})$. The output of this layer then becomes the input for the next layer. Thus, $\mathbf{x_{j+1}^h} = \mathbf{y_j^h}$.

This continues until the output layer is reached. This output layer does have a weight matrix W^o, but no activation function is applied. The output of the neural network is thus given by

$$\mathbf{y} = W^o \mathbf{x^o}, \tag{4-1-6}$$

where $\mathbf{x^o}$ is the input to the output layer. (It equals the output of the last hidden layer.)

It is interesting to note that the input signal 'flows' through the network, through all layers, until the output layer is reached. Every time, the signal is fed forward to the next layer. This is the reason why such a network is called a feedforward network.

4-2 Training neural networks

Before ANNs work, they need to be trained. That is, their weights (and biases) need to be set such that certain inputs give certain outputs. Suppose that a set of inputs $\mathbf{X}$ with corresponding desired outputs $\mathbf{D}$ is known. How does one then find the right weights? Several techniques exist. One of the simplest is the **backpropagation technique**. This technique will be examined here.

4-2-1 The underlying idea

The main idea behind backpropagation is to minimize an error function J. Almost always, the error function that is used is

$$J = \frac{1}{2} \sum_{\mathbf{X},\mathbf{D}} ||\mathbf{e}||^2, \tag{4-2-1}$$

where $\mathbf{e} = \mathbf{d} - \mathbf{y}$ is the difference between the **desired output d** and the current output $\mathbf{y}$. The summation is done over the entire set of inputs $\mathbf{X}$ and desired outputs $\mathbf{D}$. The norm in the above relation can in principle be any norm, but generally the Euclidian norm is preferred. In this report, only the Euclidian norm will be considered.

Backpropagation is a gradient descent method. It adjusts the weights in the direction of the gradient $-\partial J/\partial w$. This should reduce the cost function J. The weight update law is thus given by

$$w_{new} = w_{old} - \alpha \frac{\partial J}{\partial w}, \tag{4-2-2}$$

where α is the **learning rate**. This law can be applied for every weight w. The main difficulty of backpropagation is finding the partial derivative in the relation above. Accomplishing that is a matter of applying the derivative chain rule. Applying it for any weight w results in

$$\frac{\partial J}{\partial w} = \sum_{\mathbf{X},\mathbf{D}} \left(\frac{\partial J}{\partial \mathbf{e}} \frac{\partial \mathbf{e}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w} \right). \tag{4-2-3}$$

The first two of these three partial derivatives are easily obtained.

$$\frac{\partial J}{\partial \mathbf{e}} = \mathbf{e}^T, \tag{4-2-4}$$

$$\frac{\partial \mathbf{e}}{\partial \mathbf{y}} = -I. \tag{4-2-5}$$

It is the third derivative that is the hardest to obtain. How does the output $\mathbf{y}$ of the neural network vary with the weights w? To figure that out, the way in which the internal neural network signals (i.e., node activations and sums) depend on the output $\mathbf{y}$ needs to be considered first.

4-2-2 Derivatives with respect to node output and activation

First consider the output layer. This output layer does not have an activation; only an output $\mathbf{y^o}$. This output equals the neural network output. Thus, $\partial \mathbf{y}/\partial \mathbf{y^o} = I$. Also, it is known that $\mathbf{y} = W^o \mathbf{x^o}$. Therefore,

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x^o}} = \frac{\partial \mathbf{y}}{\partial \mathbf{y^o}} \frac{\partial \mathbf{y^o}}{\partial \mathbf{x^o}} = IW^o = W^o. \tag{4-2-6}$$

The input $\mathbf{x^o}$ to the output layer equals the output of the last hidden layer. Now consider any hidden layer j. Assume that $\partial \mathbf{y}/\partial \mathbf{y_j^h}$ is known. It is known that $\mathbf{y_j^h} = \sigma(\mathbf{z_j})$. The derivative of the NN output $\mathbf{y}$ with respect to the hidden layer node activation $\mathbf{z_j}$ now is

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z_j}} = \frac{\partial \mathbf{y}}{\partial \mathbf{y_j^h}} \frac{\partial \mathbf{y_j^h}}{\partial \mathbf{z_j}} = \frac{\partial \mathbf{y}}{\partial \mathbf{y_j^h}} \sigma'(\mathbf{z_j}), \tag{4-2-7}$$

where the derivative of the activation function is

$$\sigma'(z) = \frac{2s \exp(-sz)}{(1 + \exp(-sz))^2}. \tag{4-2-8}$$

Next, it is known that $\mathbf{z_j} = W_j^h \mathbf{x_j^h}$. With this, the derivative of the NN output $\mathbf{y}$ with respect to the hidden layer input $\mathbf{x_j^h}$ can be obtained. The result is

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x_j^h}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z_j}} \frac{\partial \mathbf{z_j}}{\partial \mathbf{x_j^h}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z_j}} W_j^h. \tag{4-2-9}$$

Since $\mathbf{x_j^h}$ equals $\mathbf{y_{j-1}^h}$ (except of course for $j = 1$), also the derivative $\partial \mathbf{y}/\partial \mathbf{y_{j-1}^h}$ is known. In this way, the derivative of the output $\mathbf{y}$ with respect to any internal signal of the neural network can be found. All it takes is recursively propagating back through the layers of the neural network.

4-2-3 Derivatives with respect to the weights

Now it is possible to find the derivative of the NN output $\mathbf{y}$ with respect to the weights. To do this, consider any node i in layer j. This node has an activation z_{ji}, where $z_{ji} = \mathbf{w_{ji}^{h}}^T \mathbf{x_j^h}$.

The derivative of the NN output $\mathbf{y}$ with respect to the weights $\mathbf{w^h_{ji}}$ is given by

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w^h_{ji}}} = \frac{\partial \mathbf{y}}{\partial z_{ji}} \frac{\partial z_{ji}}{\partial \mathbf{w^h_{ji}}} = \frac{\partial \mathbf{y}}{\partial z_{ji}} \mathbf{x^h_j}. \tag{4-2-10}$$

In this way, the partial derivative of the NN output $\mathbf{y}$ with respect to any weight w in the network can be determined. If this knowledge is combined with the update law (4-2-2) and the relations (4-2-3) through (4-2-5), one finds as update law

$$w_{new} = w_{old} + \alpha \sum_{\mathbf{X,D}} \left(\mathbf{e}^T \frac{\partial \mathbf{y}}{\partial \mathbf{w^h_{ji}}} \right). \tag{4-2-11}$$

This equation intuitively makes sense. The term $\mathbf{e}^T$ describes in which direction the output needs to be adjusted, while the partial derivative $\partial \mathbf{y} / \partial \mathbf{w^h_{ji}}$ describes how to adjust the weights to actually change this output appropriately.

4-2-4 Batch updating versus individual updating

Suppose that various input samples $\mathbf{x}$ with known desired outputs $\mathbf{d}$ are available. It is possible to find the error $\mathbf{e} = \mathbf{d} - \mathbf{y}$ for all these samples and use equation (4-2-11) to update the weights. This is called **batch updating** of the Neural Network. All input samples are considered simultaneously.

Batch updating has a few drawbacks though. Imagine that half of the input samples $\mathbf{x}$ have correct outputs $\mathbf{y}$ (that is, $\mathbf{y} \approx \mathbf{d}$), while the other half has incorrect outputs (that is, $\mathbf{y}$ differs significantly from $\mathbf{d}$). If batch updating is applied, the correct samples will not be updated – after all, the error $\mathbf{e}$ is zero – while the incorrect samples will be updated. Afterwards, the correct samples will be incorrect. Furthermore, if all of the incorrect samples require a similar update of the weights w, then the resulting update might be excessively large, quite probably leading to an overshoot. This could cause instability in the learning algorithm.

An alternative to batch updating, which suffers far less (yet still suffers a bit) from these drawbacks, is **individual updating**. In individual updating only one of the input samples is used to update the weights, again according to equation (4-2-11). Then a second input sample is used to update it again. Then a third, and so on. This continues until all samples have been used.

The additional computational cost of individual updating is relatively low. The most computationally intensive part of the entire updating algorithm is to find the partial derivatives $\partial \mathbf{y} / \partial \mathbf{w^h_{ji}}$ for all weights, and that needs to be done the same amount of times for both algorithms. The additional benefits of individual updating are, however, legion.

First of all, because the weights are adjusted much more often, the algorithm can adapt more quickly and is less sensitive to overadapting. Tests have shown that in most cases individual updating has faster convergence behavior. Secondly, applying individual updates adds a bit of randomness to the algorithm; especially if the samples are used in a random order. In many applications, including as a function approximator for a reinforcement learning algorithm, having an added randomness improves learning performance. Both these statements are also confirmed by (Schiller, 2003). Because of these added benefits, in the remainder of this report, only individual updating will be considered.

4-3 Improvements to the NN architecture

In this chapter so far, the equations have been described with which a neural network software architecture can be built. Such a software architecture has been developed for this project. During the development, several additional improvements to this architecture have also been considered and if possible incorporated, improving learning performace.

4-3-1 Normalizing input and output

When the neural network is initialized, it needs to be given weights. Which initial weights to choose can be a difficult decision. Going for zero weights is not an option, as then $\partial \mathbf{y}/\partial w$ would be zero for every weight w and no updating will occur whatsoever. Instead, weights are often initialized with a normal (Gaussian) distribution. The question remains which mean and variance this distribution should then have.

Tests have shown that a mean of zero can give an adequate learning performance. Choosing a non-zero weight has never given a significant learning performance increase, but has occasionally given a reduced learning performance. Thus a mean of zero in the random weight initialization seems to be the best option.

The variance of the distribution was harder to choose. It turned out that this strongly depends on the **Root Mean Square** (RMS) of the input and output signals. If the input signals had a big RMS, then a big variance of the nodes of the first (and only) hidden layer would improve learning performance. Similarly, if the output signals had a big RMS, then a big variance of the nodes of the output layer would improve learning performance.

One might suggest that the weights should be chosen, based on the expected RMS of the input and output signals. Instead, there is a more elegant solution. When initializing the NN, the expected RMS of the input and output signals should be defined as well. Every input and (desired) output that are subsequently passed to the NN will be normalized using this expected RMS. The weights itself are subsequently initialized with a variance of 1. This variance value has proven to give a better learning performance than nearby variance values like 3 or 0.3.

Another option (which has not been implemented in this report, but might be interesting for further research into NN learning performance) is to use an adaptive expected RMS. Every time a new input signal or desired output signal is fed to the NN, the expected RMS of the input and/or output is updated. It is expected that this will accelerate adaptation of the NN to new circumstances, especially when they involve an input and/or output RMS that varies with time.

4-3-2 Automatically varying the learning rate

The learning rate α strongly influences the speed at which the NN learns. If it's too high, learning instability might occur, while if it's too low, learning might just take too long. The trick is to set α well. An alternative, one which has not been implemented (nor tested) in this report, but which could be an interesting subject for further research, is to automatically adjust α.

In literature, adjusting the learning rate has already been applied in various ways. For example, (van Kampen, 2006) varies the learning rate based on the error E that exists between the desired NN output and actual NN output. This method has proven to work, but the effectiveness can still be improved, the reason being that the error magnitude is not always directly related to the amount of weight adjustment required.

Here an alternative will be offered to adjust the learning rate α. To see how α should be adjusted, first examine the (unlikely) case of a system with only one weight w. During one learning step, the value of w is adjusted by an amount

$$\Delta w = \alpha \sum_{\mathbf{X},\mathbf{D}} \left(\mathbf{e}^T \frac{\partial \mathbf{y}}{\partial w} \right). \tag{4-3-1}$$

After adjusting w, the value of Δw should be calculated again. Denote the first (executed) adjustment as $(\Delta w)_1$ and the second (non-executed) adjustment as $(\Delta w)_2$. If they both have the same sign, then it's an indication that w could have initially been adjusted more. In other words, $(\Delta w)_1$ could have been bigger. Thus, α should be increased. Similarly, if they have different signs, then overadjustment has taken place and α should be decreased. α should thus be adjusted by an amount

$$\Delta\alpha = \beta \frac{(\Delta w)_2}{(\Delta w)_1}, \tag{4-3-2}$$

where β is an adjustment factor for the learning rate.

Now consider an actual NN with several weights. $\Delta\mathbf{w}$ can now be seen as a vector, containing an update for all weights of the entire NN. Comparing signs of two vectors of course isn't possible. Instead, one should consider the projection of $(\Delta\mathbf{w})_2$ onto $(\Delta\mathbf{w})_1$, and whether it points in the same direction as $(\Delta\mathbf{w})_1$. An indication of this is given by

$$\rho = \frac{(\Delta\mathbf{w})_1 \cdot (\Delta\mathbf{w})_2}{||(\Delta\mathbf{w})_1|| ||(\Delta\mathbf{w})_2||}. \tag{4-3-3}$$

Subsequently, α should be adjusted by an amount

$$\Delta\alpha = \beta\rho. \tag{4-3-4}$$

As was mentioned before, this technique has not been implemented due to time constraints, so no further discussion can be held on its actual performance in a real experiment. This can be a topic for further research.

Chapter 5

Reinforcement learning with function approximators

When a reinforcement learning controller is combined with a function approximator, it becomes suitable to control continuous systems. Adding the function approximators to the RL algorithm still poses some difficulties though. How this can be done is discussed in this chapter.

First some general theories are discussed. After that, the discussion continues more in-depth on an actual way to implement neural networks into an RL controller. Concepts in this chapter have mainly originated from (Bertsekas & Tsitsiklis, 1996), (Babuška, 2010), (Sutton & Barto, 1998), (van Kampen, 2006) and own derivations.

5-1 General aspects of RL with function approximators

5-1-1 The parameter vector

Normally in RL, the value function $V(s)$ only depends on the state s. (Or similarly, $Q(s, a)$ only depends on the state s and the action a.) When using function approximators, a **parameter vector** θ is added. Now $V(\theta, s)$ depends on both θ and s. When learning, only the components of the parameter vector θ are adjusted. The function V itself remains unchanged.

During updates, changing θ will not only change the value of $V(\theta, s)$ for the current value of s, but it will also change the value of $V(\theta, s')$ for other (nearby) states s'. This **generalization** idea, of changing the value function V for multiple states simultaneously, is a fundamental aspect of RL with function approximators. It is not only a positive aspect, but (due to the infinite number of possible states) also an absolutely necessary aspect.

The important question is, how should θ be adjusted? First suppose that the goal is to find θ such that $V(\theta, s)$ approximates the value $V^\pi(s)$ of state s corresponding to a policy π. Most

supervised learning methods try to minimize a cost function J, like the **mean-squared error** (MSE),

$$J = MSE(\theta) = \sum_{s \in S} P(s) \left(V^{\pi}(s) - V(\theta, s)\right))^2, \tag{5-1-1}$$

where P is some kind of 'weighing factor,' indicating the importance of every state s. The question then is, how should θ be adjusted to reduce the MSE?

5-1-2 The learning method

A very basic way to reduce the MSE (for neural networks, but also for other function approximators) is the **gradient decent method**. Suppose that for some state s the value $V^{\pi}(s)$ is known. θ should now be adjusted in the direction of the gradient

$$-\frac{\partial \left(\left(P(s) \left(V^{\pi}(s) - V(\theta, s)\right)\right)^2 \right)}{\partial \theta}. \tag{5-1-2}$$

This results in the update law

$$\begin{aligned} \theta_{new} &= \theta_{old} - \frac{1}{2} \alpha P(s) \frac{\partial \left(V^{\pi}(s) - V(\theta_{old}, s)\right)^2}{\partial \theta_{old}} \\ &= \theta_{old} + \alpha P(s) \left(V^{\pi}(s) - V(\theta_{old}, s)\right) \frac{\partial V(\theta_{old}, s)}{\partial \theta_{old}}. \end{aligned} \tag{5-1-3}$$

In this relation, α is the **learning rate**. It indicates how much θ is adjusted in the direction of the negative gradient. Higher values of α initially may cause faster learning, but could later on result in worse convergence properties. Often it is wise to decrease α during the learning process, though this will be discussed in-depth later in this report as well.

5-1-3 Estimating the value function

Sadly, often the exact value of $V^{\pi}(s)$ is not known. Instead, only an approximation v_s of it is available. Luckily, as long as v_s is an unbiased estimate of $V^{\pi}(s)$ (so as long as $E\{v_s\} = V^{\pi}(s)$), and as long as no function approximators are applied, it can be shown that the algorithm will still always converge. If function approximators are applied, convergence is not yet guaranteed.

One such unbiased estimate is the **Monte Carlo state-value prediction**. To obtain this estimate, the agent should simply make sure it is in state s at some time k and continue to walk around, following the policy π. When it's done walking around, one can find the estimate v_s using

$$v_s = R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \dots. \tag{5-1-4}$$

This method can then also be applied for all other states s that have been visited. As long as the agent keeps walking around and visits enough states s, then the value function V will converge to V^{π}.

Alternatively, it is also possible to use **TD(λ) methods**. Assume that at some time k the parameter vector is given by θ_k. Also, at time k the state is s_k. An action a_k is selected, bringing the agent to a state s_{k+1} with immediate reward r_{k+1}. The update law now is

$$\theta_{k+1} = \theta_k + \alpha\delta_k e_k. \tag{5-1-5}$$

Here, δ_k is the **TD-error**

$$\delta_k = r_{k+1} + \gamma V(\theta_k, s_{k+1}) - V(\theta_k, s_k). \tag{5-1-6}$$

Also, e_k is a vector of eligibility traces for each component of θ_k. It satisfies

$$e_k = \gamma\lambda e_{k-1} + \frac{\partial V(\theta_k, s_k)}{\partial \theta_k}. \tag{5-1-7}$$

Sadly, for $\lambda < 1$, this method does not provide an unbiased estimate v_s. (Generally, the lower λ is, the bigger the bias is.) But although in this method V will not always converge to V^π, it occasionally learns much faster than the Monte-Carlo state-value prediction.

Note that every method can be performed similarly in case Q is used. For example, relation (5-1-5) would still work if

$$\delta_k = r_{k+1} + \gamma Q(\theta_k, s_{k+1}, a_{k+1}) - Q(\theta_k, s_k, a_k), \tag{5-1-8}$$

$$e_k = \gamma\lambda e_{k-1} + \frac{\partial Q(\theta_k, s_k, a_k)}{\partial \theta_k}, \tag{5-1-9}$$

where a_{k+1} follows from the policy π and thus equals $\pi(s_{k+1})$. The only thing that's really different is the name of the method. It is now called the **gradient-descent SARSA(λ) method**.

5-2 Setting up an RL controller with neural networks

In the previous section, it has only been discussed how to update the value function. When setting up an RL controller, it is also necessary to compute a policy. One thing that is necessary to compute the policy is knowledge on how the environment works. If that is not given, an identifier is required. With this identifier, a policy can be derived.

5-2-1 The identifier and the controller architecture

In some RL problems, no knowledge of the environment is known whatsoever. In this case, an identifier is required. The goal of the **identifier** I is to find out how the environment works. The identifier can be seen as a mapping from state s_k and action a_k to the new state s_{k+1}. Thus, $s_{k+1} \approx I(s_k, a_k)$. (Note that, for brevity, the parameter vector θ has been omitted.)

The identifier is (just like the value function) a function approximator like a neural network. As the agent walks around the environment, it continuously finds combinations (s_k, a_k, s_{k+1}). These combinations are used to train the identifier. To adequately train the identifier, it is important that it encounters a decent distribution of representative states s_k and selects a

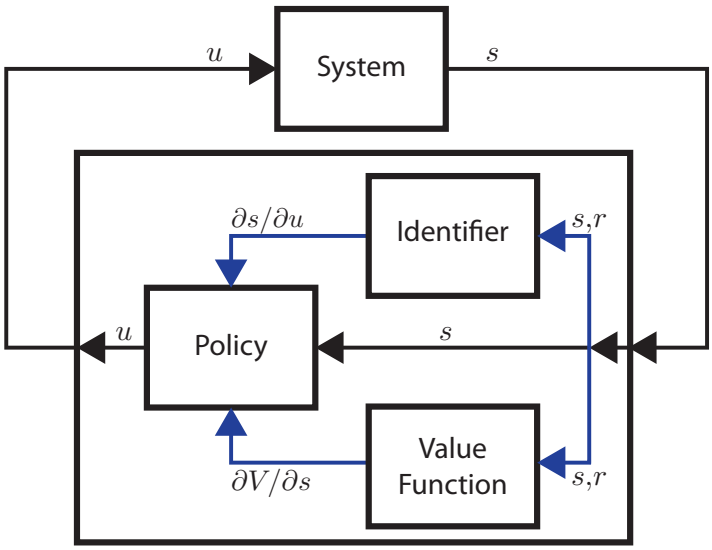

Figure 5-1: Overview of the RL controller. Knowledge of the state and the action taken is used to train the value function and the identifier. These are then subsequently used to train the policy, which provides the system with a new input.

decent distribution of representative actions a_k. If this is not the case, then the identifier will only be trained well for a small subset of the state/action space.

As the agent walks around the environment, it also finds combinations (s_k, r_{k+1}). With these combinations (and possibly additional data, like $r_{k+2}, r_{k+3}, \ldots$) it can train the value function. One may wonder: what kind of value function is best to use? V or Q? In theory it is possible to use either of them. However, if Q is used, then during the training of the RL controller, it will often be necessary to determine the optimal action

$$a = \max_{a'} Q(s, a') \tag{5-2-1}$$

for some state s. Thanks to the nonlinear nature of neural networks, the maximization in the above relation is very hard to do. It is possible to find the maximum (possibly using interval analysis, see (de Weerdt, Chu, & Mulder, 2009)) but this will computationally be very expensive, making the algorithm practically unfeasible. Thus, as value function V will be used.

Once both the identifier and the value function have been trained, the RL controller knows how the environment works (i.e., how to get from one state to another) and how valuable every state is. This is sufficient to train a policy. This entire process has been visualized in Figure 5-1.

5-2-2 Training the policy

The question remains: which policy should be used? Using a greedy policy can be troublesome. In the case of continuous actions, finding which action gives the best value is a complex nonlinear optimization problem. Instead, often a more simple approach is used: the gradient descent method.

To start the learning process, a policy π is randomly initialized. Imagine that the agent is at a state s_k. The policy π says that the agent should take an action $a_k = \pi(s_k)$. According to the identifier, this action will bring the agent to state $s_{k+1} = I(s_k, a_k)$. The goal is to maximize the total reward

$$R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \dots. \tag{5-2-2}$$

The value function can give the agent an approximation of

$$V(s_{k+1}) \approx R_{k+1} = r_{k+2} + \gamma r_{k+3} + \gamma^2 r_{k+4} + \dots. \tag{5-2-3}$$

(Again, for brevity, the parameter vector θ has been omitted from V.) Assuming that the value function is accurate, the goal of the RL controller thus is to maximize $r_{k+1} + \gamma V(s_{k+1})$. Generally, the reward r_{k+1} is derived from a known reward function r of the state reached s_{k+1} and action taken a_k. So $r_{k+1} = r(s_{k+1}, a_k)$. Thus, the goal is to maximize

$$r(I(s_k, \pi(s_k)), \pi(s_k)) + \gamma V(I(s_k, \pi(s_k))). \tag{5-2-4}$$

In the gradient descent method, the weights w (or equivalently the parameter vector θ) of the policy should be adjusted such as to increase the above quantity. The update law, for every weight w, thus becomes

$$\begin{aligned} w_{new} &= w_{old} + \alpha \frac{\partial \left(r(I(s_k, \pi(s_k)), \pi(s_k)) + \gamma V(I(s_k, \pi(s_k))) \right)}{\partial w} \\ &= w_{old} + \alpha \left(\frac{\partial r(I(s_k, \pi(s_k)), \pi(s_k))}{\partial w} + \gamma \frac{\partial V(I(s_k, \pi(s_k)))}{\partial w} \right). \end{aligned} \tag{5-2-5}$$

Note that the above relation consists of two partial derivatives. Consider the right one first. Using the chain rule, it can be expanded as

$$\frac{\partial V(I(s_k, \pi(s_k)))}{\partial w} = \frac{\partial V}{\partial s}\left(I(s_k, \pi(s_k))\right) \frac{\partial I}{\partial a}\left(s_k, \pi(s_k)\right) \frac{\partial \pi}{\partial w}\left(s_k\right). \tag{5-2-6}$$

The first derivative is the derivative of $V(s)$ with respect to s, evaluated at $I(s_k, \pi(s_k))$. The second derivative is the derivative of $I(s, a)$ with respect to a, evaluated at $(s_k, \pi(s_k))$. Finally, the third derivative is the derivative of $\pi(s)$ with respect to its own NN weights, evaluated at s_k. Note that the first two derivatives are NN input-output derivatives. How to find these has already been discussed in chapter 4 on neural networks. The third derivative is a derivative of the NN output with respect to its weights, which has also been discussed in chapter 4.

Now consider the left partial derivative of equation (5-2-5). Since the reward function r is known, as well as its derivatives $\partial r/\partial s$ and $\partial r/\partial a$, it can be expanded in an identical way. This gives

$$\begin{aligned} \frac{\partial r(I(s_k, \pi(s_k)), \pi(s_k))}{\partial w} &= \frac{\partial r}{\partial s}\left(I(s_k, \pi(s_k)), \pi(s_k)\right) \frac{\partial I}{\partial a}\left(s_k, \pi(s_k)\right) \frac{\partial \pi}{\partial w}\left(s_k\right) \\ &\quad + \frac{\partial r}{\partial a}\left(I(s_k, \pi(s_k)), \pi(s_k)\right) \frac{\partial \pi}{\partial w}\left(s_k\right). \end{aligned} \tag{5-2-7}$$

The full weight update law can now be condensed into

$$\begin{aligned} w_{new} &= w_{old} + \alpha \left(\left(\frac{\partial r}{\partial s}\left(I(s_k, \pi(s_k)), \pi(s_k)\right) + \gamma \frac{\partial V}{\partial s}\left(I(s_k, \pi(s_k))\right) \right) \frac{\partial I}{\partial a}\left(s_k, \pi(s_k)\right) \right. \\ &\quad \left. + \frac{\partial r}{\partial a}\left(I(s_k, \pi(s_k)), \pi(s_k)\right) \right) \frac{\partial \pi}{\partial w}\left(s_k\right). \end{aligned} \tag{5-2-8}$$

With this relation, the policy can be updated according to the gradient descent method. Consequentially, it will select actions that result in higher values.

It is important to note that this whole process of improvement is cyclic. First the value function needs to be trained for a given policy. Then the policy needs to be improved, based on the value function. Then the value function needs to be adjusted for the new policy. Then the policy is improved again. And so on. One can wonder: will this ever converge to a stationary situation? And if so, will this stationary situation have the optimal policy or a sub-optimal policy? If it is the optimal policy, can it also be proven that the optimal policy will always be found? Those are interesting questions that will be further discussed in later chapters.

5-3 Making the controller continuous in time

This chapter so far has shown how to make an RL controller that can deal with both continuous states and continuous actions. It is however not yet continuous in time. There will always be time steps Δt. In each time step, the action a that is used will be constant, and it will not change until the next time step commences. This can be seen as a downside: if the time steps are relatively big, then the achievable control accuracy is limited.

Is there an easy way to get around this problem? Sadly, the answer is no. On digital computers, discrete time steps will always be required. To get rid of time steps, a system with analog hardware needs to be designed, but that is outside of the scope of this project. The solution to this problem is to vary the time step Δt to find which one works best. It needs to be small enough to prevent significant effects of this problem, yet still big enough to get a good computational efficiency.

It is here that a problem arises. The neural networks of the RL controller are all trained for a specific time step Δt. Changing the time step will change the required NN outputs, making the NNs obsolete. But is there a way to solve this problem? This time the answer is yes.

5-3-1 The identifier

First consider the identifier. Previously, the identifier's output was the next state s_{k+1}. In a continuous system, it would be more appropriate for the identifier to output the state derivative $\dot{s}$. Once this is know, the next state will be given by $s_{k+1} = s_k + \dot{s}\Delta t$. As long as the time step Δt is kept sufficiently small, it may be varied at will without getting any significant deviations.

5-3-2 The value function

Next, consider the value function. As has been mentioned before, the value function is the estimation of the weighted sum of all future rewards

$$V(s) \approx R_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{k+i+1}. \tag{5-3-1}$$

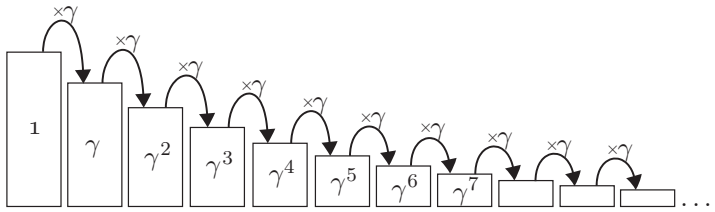

Figure 5-2: Weighing of the reward function in conventional constant-timestep RL. The weight of every time step is a factor γ times the previous time step.

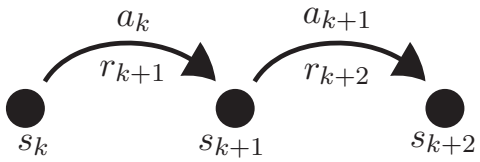

Figure 5-3: Transitioning of states in reinforcement learning. States are points in time, while actions are taken over time and rewards are obtained over time.

These weights have been visualized in Figure 5-2. The problem is that, if the time step Δt is decreased, then the RL agent will encounter many more states within the same time span. The result is that it will think much more short-term. This should be compensated for.

The solution here is to realize that rewards are always given over a certain time span, as is shown in Figure 5-3. A reward r_{k+1} is not given simply for reaching the state s_{k+1}. It is the reward that has resulted from the entire process of selecting action a_k to reach state s_{k+1} from state s_k. It concerns the entire time step between time t_k and time t_{k+1}. This gives rise to the idea of replacing the reward weighing by a continuous weight function $w_r(t)$.

Using this idea, the expression for the total reward can be adjusted. The weight for the time step from s_{k+i} to s_{k+i+1} is now not simply γ^i. It becomes the integral of the weight function $w_r(t)$ over this time step, where time is measured starting from t_k. So the weight is

$$\int_{t_{k+i}}^{t_{k+i+1}} w_r(t - t_k)\, dt. \tag{5-3-2}$$

If this is inserted into the relation for the total reward at time t_k, the result is

$$R_k = \sum_{i=0}^{\infty} \left(\int_{t_{k+i}}^{t_{k+i+1}} w_r(t - t_k)\, dt \right) r_{k+i+1}. \tag{5-3-3}$$

A useful and intuitive weight function $w_r(t)$ would be the decreasing exponential

$$w_r(t) = \gamma^t\, dt, \tag{5-3-4}$$

with $0 < \gamma < 1$. The integral of this function equals

$$W_r(a, b) = \int_a^b \gamma^t\, dt = \frac{\gamma^b - \gamma^a}{\ln \gamma}. \tag{5-3-5}$$

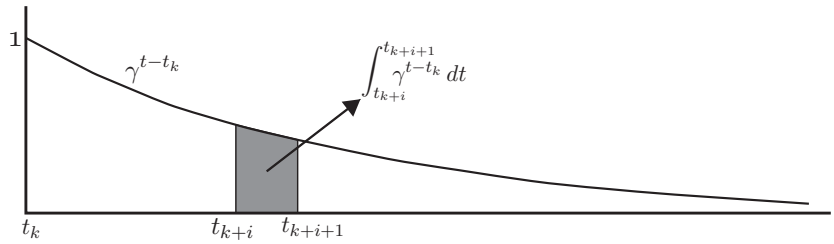

Figure 5-4: Weighing of the reward function in variable-timestep RL. The reward weight now is continuous. In this case, it is an exponentially decreasing function of time.

This turns the relation for the total reward into

$$R_k = \sum_{i=0}^{\infty} \left(\frac{\gamma^{(t_{k+i+1}-t_k)} - \gamma^{(t_{k+i}-t_k)}}{\ln \gamma} \right) r_{k+i+1}. \tag{5-3-6}$$

This method of weighing rewards is visualized in Figure 5-4.

Using this technique will also change the update law in a TD method. Previously, the TD error was defined as

$$\delta_k = r_{k+1} + \gamma V(s_{k+1}) - V(s_k). \tag{5-3-7}$$

In this expression, the reward r_{k+1} has a weight of 1. In the new method this is different. To find the weight, the weight function w_r should be integrated from 0 to $t_{k+1} - t_k$. r_{k+1} is thus weighed by

$$W_r(0, t_{k+1} - t_k) = \frac{\gamma^{t_{k+1}-t_k} - 1}{\ln \gamma}. \tag{5-3-8}$$

The term $V(s_k)$ has already been weighed appropriately, with a total weight of $W_r(0, \infty) = -1/\ln(\gamma)$. (Remember that $V(s_k)$ is an estimate of R_k.) The same holds for the term $V(s_{k+1})$. However, in reality this latter term should be weighed with a total weight of $W_r(t_{k+1} - t_k, \infty)$. This should be compensated for. The TD error thus becomes

$$\delta_k = W_r(0, t_{k+1} - t_k) r_{k+1} + \frac{W_r(t_{k+1} - t_k, \infty)}{W_r(0, \infty)} V(s_{k+1}) - V(s_k). \tag{5-3-9}$$

Implementing relations (5-3-4) and (5-3-5) turns the TD error into

$$\delta_k = \frac{\gamma^{t_{k+1}-t_k} - 1}{\ln \gamma} r_{k+1} + \gamma^{t_{k+1}-t_k} V(s_{k+1}) - V(s_k). \tag{5-3-10}$$

5-3-3 The policy

Finally the policy needs to be made independent of the time step. Initially one might think that the policy is strongly affected by the time step. After all, it is very different to apply an action a for only a brief time step Δt or to apply the same action a for a very long time step Δt. But this subsection will show that that is actually not the case.

First, examine the weight update law (5-2-8). This law should be updated using the right weighing for all parameters. Applying relations (5-3-4) and (5-3-5) right away to the policy

weight update law results in

$$w_{new} = w_{old} + \alpha \left(\left(\frac{\gamma^{t_{k+1}-t_k} - 1}{\ln \gamma} \frac{\partial r}{\partial s} \left(I(s_k, \pi(s_k)), \pi(s_k) \right) + \gamma^{t_{k+1}-t_k} \frac{\partial V}{\partial s} \left(I(s_k, \pi(s_k)) \right) \right) \right.$$
$$\left. \frac{\partial I}{\partial a} \left(s_k, \pi(s_k) \right) + \frac{\gamma^{t_{k+1}-t_k} - 1}{\ln \gamma} \frac{\partial r}{\partial a} \left(I(s_k, \pi(s_k)), \pi(s_k) \right) \right) \frac{\partial \pi}{\partial w} \left(s_k \right). \tag{5-3-11}$$

Now the question arises: would variations in $\Delta t = t_{k+1} - t_k$ change this update law? And if so, how?

Of course, if Δt is very big, or equivalently if γ is very small, there will be variations. In most cases though, $\Delta t \approx 0$ and $\gamma \approx 1$. If this holds, then there are only a few terms that vary with Δt. $\partial r / \partial s$ does not, and nor do $\partial V / \partial s$, $\partial r / \partial a$ and $\partial \pi / \partial w$. However, both the term $\partial I / \partial a$ and the weighing factors that contain Δt do vary with Δt. The way in which they do can best be described using a first-order Taylor expansion about $\Delta t = 0$. This gives

$$\frac{\gamma^{\Delta t} - 1}{\ln \gamma} \approx \gamma^{\Delta t} \Delta t \approx \Delta t, \tag{5-3-12}$$
$$\gamma^{\Delta t} \approx \ln(\gamma) \Delta t \approx 0, \tag{5-3-13}$$
$$\frac{\partial I}{\partial a} \approx \frac{\partial \dot{s}}{\partial a} \Delta t. \tag{5-3-14}$$

(To see why this last part holds, recall that the identifier I returns the state s_{k+1}, given the state s_k and the action a_k. And if the time step Δt is small, the state derivative $\dot{s}$ can be assumed constant between t_k and t_{k+1}, thus approximating s_{k+1} as $s_{k+1} = s_k + \dot{s} \Delta t$. Taking the partial derivative with respect to the action a directly results in the above relation.)

Based on these results, two things can be noted when the time factor Δt is halved. First of all, the importance of the reward r_{k+1} will also be halved. Relatively more attention will be paid to the actual value of the state s_{k+1} that is reached. Secondly, the amount with which the weights of the policy will be updated will also be halved. However, these are the only effects that a changing time step Δt would have on the policy. And if the above update is simply executed twice (because there will be double the number of time steps), then the effects will mostly be negated. Thus, it seems that a varying time step Δt does not significantly influence the way in which the policy is updated.

With the expansion of reinforcement learning described in this section, it is thus possible to make an RL controller in which the time step can be freely varied without any complications. This not only removes the constraint of a constant time step, but it also creates various opportunities. An example of such an opportunity is a controller which can decide for itself which time step it should take. However, such a controller would be outside of the scope of this project, and thus remains merely a suggestion for further research.

Chapter 6

Interval analysis

Interval Analysis (IA) has proven to be a useful technique in finding bounds on parameters. It has uses in setting up proofs for mathematical bounds, as well as in optimization algorithms. As such, it could prove to be useful in finding convergence proofs for RL learning algorithms. This chapter explains the basic theories and ideas behind interval analysis. It is a medium-level summary of (Moore, Baker Kearfott, & Cloud, 2009) and is written to provide anyone new to the field of IA with a quick yet easy introduction. For various interesting optimization applications of IA, the reader is referred to (van Kampen, 2010).

Initially some basic interval properties are discussed. The summary then continues with discussing interval sequences and interval functions. Later on more advanced techniques are discussed, like how to use intervals to find good bounds for functions, how interval matrices work and how intervals can be used to bound differential equation solutions.

6-1 Interval basics – Notations, definitions and operations

What are intervals, what properties do they have and what can be done with them? Those questions are discussed in this first section.

6-1-1 Introduction to intervals

An **interval** $[a, b]$ is a special type of set. It contains all real numbers $x \in \mathbb{R}$ for which $a \leq x \leq b$. If a number x falls in an interval $[a, b]$, then one can write $x \in [a, b]$. The number a is called the **lower bound** of the interval, while b is the **upper bound**.

Consider the equation

$$x^2 - c = 0, \tag{6-1-1}$$

where the value of c is unknown. It is only known that $c \in [2, 3]$. What does this tell about x? In this simple case, it can right away be seen that $x \in [\sqrt{2}, \sqrt{3}]$. That is, $[\sqrt{2}, \sqrt{3}]$ provides an **interval enclosure** for the solution of x. It is also the **most narrow interval enclosure**

possible. However, for more complicated equations, finding a narrow interval enclosure isn't so easy. In this summary, several methods will be investigated with which interval enclosures can be found and subsequently made as narrow as possible.

6-1-2 Outward rounding and interval hulls

Solving equations with interval analysis is usually performed by computers, and computers aren't so good with numbers like $\sqrt{3}$. They would use a rounded number, like 1.73. But if a computer would use 1.73 as upper bound for our previous problem, things could go horribly wrong: the solution $x = 1.732$ would be ignored! For this reason, when rounding intervals, it's always appropriate to play it safe: round the lower bound downwards and the upper bound upwards. Of course it evidently holds that $x \in [1.41, 1.74]$. This very important idea is called **outward rounding**.

Note that not all sets can be expressed as intervals. For example, the set of all numbers x with either $0 \leq x \leq 1$ or $2 \leq x \leq 3$ contains a gap, and can thus not be expressed as an interval. However, all possible values x can be bound using an **interval hull**. It is then simply said that $x \in [0, 3]$.

6-1-3 Notations and definitions

From now on, intervals will be denoted with capital letters, like X. (So for example, $X = [1.41, 1.74]$.) The two endpoints of an interval are denoted by $\underline{X}$ and $\overline{X}$. Thus, $X = [\underline{X}, \overline{X}]$. Now several properties will be listed which intervals might have.

- If an exact number x falls in an interval X, then $x \in X$.
- If $\underline{X} = \overline{X}$ (and X is thus an exact number), then X is called **degenerate**. Degenerate intervals are denoted by lower case letters, like x.
- Two intervals are said to be **equal** if both $\underline{X} = \underline{Y}$ and $\overline{X} = \overline{Y}$. This is written as $X = Y$.
- One can write $X < Y$ (X is **lower than** than Y) if all numbers $x \in X$ and $y \in Y$ satisfy $x < y$. So $X < Y$ if $\overline{X} < \underline{Y}$. Similarly, $X > Y$ if $\underline{X} > \overline{Y}$. (Note that, for two intervals X and Y, it is possible that neither $X = Y$, nor $X < Y$ or $X > Y$. An example of this is $X = [2, 4]$ and $Y = [3, 5]$.)
- An interval X is called **positive** if $X > 0$ (or equivalently $\underline{X} > 0$). Similarly, it is **negative** if $X < 0$.
- One may write $X \subseteq Y$ if all numbers $x \in X$ also satisfy $x \in Y$. This is the case if $\underline{X} \geq \underline{Y}$ and $\overline{X} \leq \overline{Y}$.
- The **absolute value** of an interval X is defined as $|X| = \max(|\underline{X}|, |\overline{X}|)$. So for every number $x \in X$, it holds that $|x| \leq |X|$.
- The **width** of an interval X is defined as $w(X) = \overline{X} - \underline{X}$.

- The **midpoint** of an interval X is defined as

$$m(X) = \frac{1}{2}(\underline{X} + \overline{X}). \tag{6-1-2}$$

  Note that any interval X can now be written as $\left[m(X) - \frac{1}{2}w(X), m(X) + \frac{1}{2}w(X)\right]$.

- An interval is **symmetric** if $\underline{X} = -\overline{X}$. That is, if $m(X) = 0$.

6-1-4 Operations

There are several operations that can be performed on intervals X and Y. The most important ones will be examined here.

- The **intersection** $X \cap Y$ of two intervals x and Y consists of all numbers z such that both $z \in X$ and $z \in Y$. If either $X > Y$ or $X < Y$, then $X \cap Y = \emptyset$. That is, the intersection equals the **empty set**. Otherwise, it holds that

$$X \cap Y = \left[\max(\underline{X}, \underline{Y}), \min(\overline{X}, \overline{Y})\right]. \tag{6-1-3}$$

- The **union** $X \cup Y$ of two intervals X and Y consists of all numbers z such that $z \in X$ or $z \in Y$ or both. If neither $X > Y$ nor $X < Y$, then it is easily found that

$$X \cup Y = \left[\min(\underline{X}, \underline{Y}), \max(\overline{X}, \overline{Y})\right]. \tag{6-1-4}$$

  However, if $X < Y$ or $X > Y$, then the union of X and Y can't be described with one simple interval. (There will be a gap between the intervals X and Y.) It is possible to use multiple intervals to describe the union, but as the number of intervals grows, it could significantly complicate computations. Instead, often the interval hull is used, resulting in

$$X \underline{\cup} Y = \left[\min(\underline{X}, \underline{Y}), \max(\overline{X}, \overline{Y})\right]. \tag{6-1-5}$$

  Note that $X \cup Y \subseteq X \underline{\cup} Y$.

- Arithmetic operations (like $+$, $-$, $\cdot$ and $/$) can also be applied to intervals. Let's denote $\odot$ as any such type of operation. Then it is said that $X \odot Y$ equals the set of all numbers z such that there are $x \in X$ and $y \in Y$ with $z = x \odot y$. In set notation, this becomes

$$X \odot Y = \{x \odot y : x \in X, y \in Y\}. \tag{6-1-6}$$

- An example of the above item is the **sum** $X + Y$ of two intervals X and Y. It can be determined that

$$X + Y = [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}]. \tag{6-1-7}$$

- The **difference** $X - Y$ is given by

$$X - Y = X + (-Y) = [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}]. \tag{6-1-8}$$

  Note that $-Y = [-\overline{Y}, -\underline{Y}]$. (So next to adding a minus sign, also the upper and lower bounds are interchanged.)

- The **product** $X \cdot Y$ is given by

$$X \cdot Y = [\min(S), \max(S)], \quad \text{where } S = \left\{\underline{X}\underline{Y}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{X}\overline{Y}\right\}. \tag{6-1-9}$$

  Often, the dot '$\cdot$' is not written down. So XY also denotes the product of X and Y.

- The **division** X/Y is given by

$$X/Y = X \cdot (1/Y) = [\underline{X}, \overline{X}] \cdot \left[1/\overline{Y}, 1/\underline{Y}\right]. \tag{6-1-10}$$

  Note that the division operator doesn't work when $0 \in Y$.

- All above operations also work in case a constant c (i.e. a degenerate intervals) is used. For example, $X + c = [\underline{X} + c, \overline{X} + c]$. So interval arithmetics is simply an expansion of normal arithmetics.

6-2 Interval basics – Arithmetics properties and vectors

In this section, the discussion on the basics of interval analysis and interval arythmetics is continued, but now some more in-depth properties will be discussed.

6-2-1 An example to show interval dependency

Consider the problem of finding all values z satisfying

$$z = \frac{x}{1+x}, \tag{6-2-1}$$

where $x \in X = [1, 2]$. To do this, interval arithmetics can be used. First all terms x are replaced by the corresponding interval X. So,

$$Z = \frac{X}{1+X}. \tag{6-2-2}$$

Then the basic interval arithmetics rules are applied and things are worked out further. This gives, step by step,

$$1 + X = [2, 3], \tag{6-2-3}$$

$$\frac{1}{1+X} = [1/3, 1/2] \tag{6-2-4}$$

$$Z_1 = \frac{X}{1+X} = [1, 2][1/2, 1/3] = [1/3, 1]. \tag{6-2-5}$$

Alternatively, one can also use

$$Z = \frac{1}{1+\frac{1}{X}}. \tag{6-2-6}$$

This time it is found that

$$\frac{1}{X} = [1/2, 1] \tag{6-2-7}$$

$$1 + \frac{1}{X} = [3/2, 2] \tag{6-2-8}$$

$$Z_2 = \frac{1}{1 + \frac{1}{X}} = [1/2, 2/3]. \tag{6-2-9}$$

But those two intervals are different! Which one is correct? The answer is: they're both correct, but the second interval Z_2 is much narrower, and thus more useful.

The question remains, why do the results differ? The reason lies in **interval dependency**. In the first calculation, the term X appeared twice in the equation. If X would be replaced by a random number $x \in X$, then the x for the first occurrence of X has to be the same as the x for the second occurrence of X. But when applying interval arithmetics, this was not taken into account. In fact, what was calculated was

$$Z_1 = \frac{X}{1 + Y}, \quad \text{with } X = [1, 2] \text{ and } Y = [1, 2],\ X \text{ and } Y \text{ independent.} \tag{6-2-10}$$

For this problem, Z_1 is indeed the narrowest interval possible. However, for the actual problem it must hold that $x = y$, and then Z_2 is the narrowest interval possible. This interval dependency is a very important (and sometimes confusing) aspect of interval analysis.

6-2-2 Properties of addition and multiplication

Next up in the discussion are addition and multiplication. What kind of properties do they have? To start with, are these operations **commutative** and **associative**? It turns out that they are. In other words, it holds that

$$X + Y = Y + X, \qquad X + (Y + Z) = (X + Y) + Z, \tag{6-2-11}$$

$$XY = YX, \qquad X(YZ) = (XY)Z. \tag{6-2-12}$$

However, the operations are not **distributive**. That is, it generally does not hold that

$$X(Y + Z) = XY + XZ. \tag{6-2-13}$$

The reason for this is that the term X appears twice on the right hand side, so interval dependency again occurs. The interval on the right hand side will thus be broader than the one on the left hand side. It now follows that

$$X(Y + Z) \subseteq XY + XZ. \tag{6-2-14}$$

This property is called **subdistributivity**.

And what about **identity elements**? Do the operations have them? Yes, they do. The identity elements are respectively 0 and 1. So for any interval X it holds that

$$0 + X = X + 0 = X \quad \text{and} \quad 1 \cdot X = X \cdot 1 = X. \tag{6-2-15}$$

However, in IA, there is no such thing as an **inverse**. In fact,

$$X + (-X) = \left[\underline{X} - \overline{X}, \overline{X} - \underline{X}\right] \neq 0. \tag{6-2-16}$$

(That is, unless X is degenerate.) A similar thing holds for multiplication.

Finally, there is the **cancellation law**. This law holds for addition but not for multiplication. That is,

$$X + Z = Y + Z \Rightarrow X = Y \quad \text{but not} \quad XZ = YZ \Rightarrow X = Y. \tag{6-2-17}$$

6-2-3 Interval vectors

As a final part in this chapter, **interval vectors** are considered. An n-dimensional interval vector is a vector $X = (X_1, \ldots, X_n)$ of intervals. Such vectors are also denoted using capital letters, like X. In fact, from now on when writing X, always a (possibly one-dimensional) interval vector is considered.

Of course interval vectors have various properties as well.

- It is said that a vector $x = (x_1, \ldots, x_n)$ of normal (degenerate) numbers satisfies $x \in X$ if for every element i it holds that $x_i \in X_i$. So, while a one-dimensional interval describes a region on the number line, a two-dimensional interval vector describes a rectangular area in the two-dimensional plane. Similarly, an n-dimensional interval vector describes a box in n-dimensional space.
- If an interval vector X has at least one element X_i being the empty set $\emptyset$, then it is said that $X = \emptyset$. That is, X itself is empty.
- It is said that $X \subseteq Y$ if and only if for all elements X_i and Y_i it holds that $X_i \subseteq Y_i$. (So the n-dimensional box Y completely encompasses the box X.)
- The **intersection** of two interval vectors X and Y can be found by intersecting each individual element. So,

  $$X \cap Y = (X_1 \cap Y_1, \ldots, X_n \cap Y_n). \tag{6-2-18}$$

  This works identically for the **union** of two interval vectors.
- The scalar **width** $w(X)$ of an interval vector X is the largest of the widths of any of the elements X_i. So, $w(X) = \max w(X_i)$.
- The **midpoint** $m(X)$ of an interval vector X is defined as the vector $m(X) = (m(X_1), \ldots, m(X_n))$.
- The **norm** $||X||$ of an interval vector X is defined as the largest of the absolute values $|X_i|$ of any of the elements X_i. So, $||X|| = \max |X_i|$.
- The **inner product** $P = X \cdot Y$ of two vectors X and Y consists of the sum of the products of the individual elements. So,

  $$P = XY = X_1Y_1 + \ldots + X_nY_n. \tag{6-2-19}$$

Next to interval vectors, it is also possible to build interval matrices by putting intervals in a matrix form. This will be treated in a later section.

6-3 Interval sequences

Working with intervals often requires a lot of iterations, which results in a whole sequence of intervals. In this chapter, such interval sequences are going to be looked at more closely.

6-3-1 A distance metric for interval analysis

In many parts of calculus, including sequence analysis, there is the need for some measure of 'distance' between numbers. This measure is called a **distance metric** d and is always defined on some **metric space** S. For all elements $x, y \in S$, the metric d must satisfy

$$\begin{aligned} d(x,y) &= 0 \text{ if and only if } x = y, && \text{(6-3-1)} \\ d(x,y) &= d(y,x), && \text{(6-3-2)} \\ d(x,y) &\leq d(x,z) + d(z,y) \text{ for any } z. && \text{(6-3-3)} \end{aligned}$$

For real numbers a common metric is $d(x,y) = |x - y|$. This won't work so well for intervals though, so in the field of interval analysis the most commonly used distance metric is

$$d(X,Y) = \max\left(|\underline{X} - \underline{Y}|, |\overline{X} - \overline{Y}|\right). \tag{6-3-4}$$

In the rest of this summary, this is the distance metric that will be used.

6-3-2 Convergence in interval analysis

Examine a sequence of numbers $\{x_k\}$. The sequence is called **convergent** if there is a real number x^* such that, for every ε, there is a natural number $N = N(\varepsilon)$ with

$$|x_k - x^*| < \varepsilon \tag{6-3-5}$$

whenever $k > N$. In this case, x^* is called the **limit** of the sequence $\{x_k\}$.

It is possible to translate this definition to the world interval analysis. Consider an **interval sequence** $\{X_k\}$. Instead of using the distance metric $d(x,y) = |x - y|$, one now uses the distance metric of the previous paragraph. So, it is said that an interval sequence $\{X_k\}$ is called **convergent** if there is an interval X^* such that, for every ε, there is a natural number $N = N(\varepsilon)$ with

$$d(X_k - X^*) < \varepsilon \tag{6-3-6}$$

whenever $k > N$. In this case, X^* is the **limit interval** of the sequence $\{X_k\}$. It means that $\underline{X}_k$ converges to $\underline{X}^*$ and $\overline{X}_k$ converges to $\overline{X}^*$.

6-3-3 Nesting

Again consider an interval sequence $\{X_k\}$. Such a sequence is called **nested** if $X_{k+1} \subseteq X_k$. A nested sequence always convergences.

Consider a progress that iteratively solves an equation using interval analysis. During every step k, it is known that the solution x lies in the interval X_k. However, the sequence $\{X_k\}$ does

not necessarily converge. To solve this problem, a nice option is to apply **nesting**. Simply take the intersection Y_k of all intervals X_i (with $i = 1, \ldots, k$). Recursively, this becomes

$$Y_k = X_k \cap Y_{k-1}, \tag{6-3-7}$$

with $Y_1 = X_1$. The sequence Y_k is the nested version of X_k, so it converges. And since $x \in X_k$ for all k, it must also hold that $x \in Y_k$ for all k. This thus results in a converging bound on the solution for x.

A sequence $\{X_k\}$ is said to have **finite convergence** if there is a positive integer K such that $X_k = X_K$ for $k \geq K$. In other words, after the K^{th} interval, X_k stops changing. In analytical interval analysis, finite convergence rarely occurs. However, in computational interval analysis finite convergence always occurs. This is because, within the precision of the computer, X_k can only have a finite amount of values. A converging series will therefore always converge to one of these values.

6-4 Interval functions

Consider a function $f(x)$. (An example is $f_1(x) = e^x$ with $x \in \mathbb{R}$, or $f_2(x) = x_1 + x_2$, with $x = (x_1, x_2) \in \mathbb{R}^2$.) Officially, a function always consists of both a formula and a domain, and normally intervals aren't part of this domain. So to apply interval analysis to functions, it is first necessary to extend their domains to include intervals. There are multiple ways to do this. In this chapter, two important ways will be considered. After that, the relationship between these two ways will be examined.

6-4-1 The united extension and the interval extension

The first method is called the **united extension**. To find it, all possible values of $f(x)$ (with $x \in X$) are examined and then assembled (united) in one set. The resulting set is denoted by $f(X)$ and is officially defined as

$$f(X) = \{f(x_1, \ldots, x_n) : x_1 \in X_1, \ldots, x_n \in X_n\} = \{f(x) : x \in X\}. \tag{6-4-1}$$

It is important to note that the set $f(X)$ cannot always be described by an interval. For example, when f contains discontinuities, then the resulting set may contain gaps. Nevertheless, $f(X)$ is defined as the smallest interval that contains all possible values of $f(x)$ with $x \in X$.

To find $f(X)$, it is necessary to look at all possible $x \in X$. If X contains real numbers, this is often impossible and it is thus not possible to find $f(X)$. Luckily, there a few cases in which $f(X)$ can be found quite easily. Suppose that $f(x)$ is increasing in x. (Or, if x is a vector, that $f(x)$ is increasing in all parameters x_i.) It then holds that

$$f(X) = \left[f(\underline{X}), f(\overline{X})\right]. \tag{6-4-2}$$

This relation can be applied to functions like e^x or $\ln(x)$, as well as parts of functions, like $\sin(x)$ (with $-\pi/2 \leq x \leq \pi/2$) or x^y (with $x > 0$ and $y > 0$). Similarly, if $f(x)$ is decreasing in x, then

$$f(X) = \left[f(\overline{X}), f(\underline{X})\right]. \tag{6-4-3}$$

Another way to extend a function is to use the **(natural) interval extension**, denoted by $F(X)$. To find it, start with the function $f(x)$. Then replace every term x_i by the corresponding interval X_i. Finally, use interval arithmetics to evaluate the function. For example, if

$$f(x_1, x_2) = \frac{x_1}{1 + x_2}, \quad \text{then} \quad F(X_1, X_2) = \frac{X_1}{1 + X_2} = \frac{\left[\underline{X}_1, \overline{X}_1\right]}{1 + \left[\underline{X}_2, \overline{X}_2\right]}, \tag{6-4-4}$$

where the resulting interval equation can be worked out further. In general, interval extensions are a lot easier to work with than united extensions. The downside is that interval extensions don't always provide a very narrow bound.

6-4-2 The fundamental theorem of interval analysis

Now it is time to discuss the fundamental theorem of interval analysis, but before that can be done, first some extra definitions need to be considered.

- A function $f(x)$ is a **rational function** if its values are defined by a specific finite sequence of the interval arithmetic operations $+$, $-$, $\cdot$ and $/$. In other words, it is possible to find the value of $f(x)$ by only adding, subtracting, multiplying and/or dividing.
- If an interval-valued function $F(X)$ is a rational function, it is called a **rational interval function**. An example of such a function is $F_1(X) = [-1, 1](X + [1, 2])$. On the other hand, $F_2(X) = \frac{1}{2}X + \frac{1}{2}m(X)$ is not a rational interval function due to the presence of $m(X)$.
- A function $f(x)$ is an **elementary function** if its values are defined by a specific finite sequence of interval arithmetic operations and elementary unary functions. (A **unary function** is a function with only one parameter. **Elementary unary functions** include (among others) exponentials, logarithms and n^{th} roots.) Note that all rational functions are elementary functions.
- If an elementary function $f(x)$ takes as input n variables (so the vector x is of size n), then it is said that f is in the class $FC_n(X_0)$ (written as $f \in FC_n(X_0)$).
- An interval-valued function $F(X)$ is said to be inclusion isotonic if

  $$X \subseteq Y \Rightarrow F(X) \subseteq F(Y). \tag{6-4-5}$$

  United extensions are always inclusion isotonic. The same holds for elementary functions. An example of a function that is not inclusion isotonic is $F_2(X) = \frac{1}{2}X + \frac{1}{2}m(X)$.

With the help of these properties, it is possible to discuss the **fundamental theorem of interval analysis**. This theorem states that, if F is an inclusion isotonic interval extension of a function f, then

$$f(X) \subseteq F(X). \tag{6-4-6}$$

In other words, the interval extension always encompasses the united extension. And, when $f(x)$ is an elementary function in which every parameter x_i occurs only once (i.e. there is no interval dependency), then it even holds that $f(X) = F(X)$. Sadly, in practice this is rarely ever the case.

6-4-3 Excess width and refinement

Consider an inclusion isotonic interval extension $F(X)$ of some function $f(x)$. It is known that $f(X) \subseteq F(X)$. In other words, $F(X)$ provides some sort of bound for $f(X)$, but often it is too big. In fact, the **bounding error** E is defined such that

$$f(X) = F(X) + E. \tag{6-4-7}$$

(Note that this does not imply that $E = f(X) - F(X)$.) To know how good the bound $F(X)$ is, one can look at the width of E. This width is called the **excess width** and can be found using

$$w(E(X)) = w(f(X)) - w(F(X)). \tag{6-4-8}$$

Suppose the goal is to find an as accurate bound on $f(X)$ as possible. (The excess width should be minimized.) One way to do that is to use **refinement**. The first thing that is then done is to apply a **uniform subdivision** to the interval vector X. Every element X_i is split up into N equally wide subintervals $X_{i,j}$. So,

$$X_{i,j} = \left[\underline{X}_i + \frac{j-1}{N} w(X_i), \underline{X}_i + \frac{j}{N} w(X_i)\right]. \tag{6-4-9}$$

Note that this gives a total of N^n subintervals, and if they're all put together again, the original interval X is once more obtained.

Now it is possible to find the **refinement** $F_{(N)}(X)$ over X. This is done using

$$F_{(N)}(X) = \bigcup_{j_i=1}^{N} F(X_{1,j_1}, \ldots, X_{n,j_n}). \tag{6-4-10}$$

In other words, each of the N^n subintervals is inserted into $F(X)$ and then all the resulting outcomes are merged together. The resulting set $F_{(N)}(X)$ has a lower excess width than the original set $F(X)$. In fact, define the **refinement bounding error** E_N such that

$$F_{(N)}(X) = f(X) + E_N(X). \tag{6-4-11}$$

The **refined excess width** $w(E_N(X))$ now turns out to be inversely proportional to N. In other words, there is a constant K such that $w(E_N(X)) \leq Kw(X)/N$. So if N becomes twice as big, the excess width becomes (roughly) twice as small.

6-5 Narrowing the interval extension $F(X)$

Next, consider an elementary function $f(x)$, its united extension $f(X)$ and its interval extension $F(X)$ on some interval X. If every term x_i appears only once in the relation for $f(x)$, and there is thus no interval dependency, then $f(X) = F(X)$, but if there is interval dependency, it only holds that $f(x) \subseteq F(X)$. The question then is, how can someone narrow down $F(X)$? One method that can make $F(X)$ more narrow (refinement) has already been discussed. This section will examine various more such methods.

6-5-1 The centered form

The first method that well be looked at is called the **centered form** (CF). The idea behind it is to reduce the effects of interval dependency. When applying it, first c is defined as $c = m(X)$. Then x is substituted by $y + c$ (or equivalently, x_i by $y_i + c_i$) and $g(y)$ is found using

$$g(y) = f(y + c) - f(c). \tag{6-5-1}$$

When $f(x)$ is a polynomial, then it is possible to write $g(y)$ in the form

$$g(y_1, \ldots, y_n) = y_1 h_1(y_1, \ldots, y_n) + \ldots + y_n h_n(y_1, \ldots, y_n). \tag{6-5-2}$$

Thus, $f(x)$ can be written as

$$f(x) = f(c) + y_1 h_1(y) + \ldots + y_n h_n(y). \tag{6-5-3}$$

(If $f(x)$ is not a polynomial, simply write as many terms as possible in the form $y_i h_i(y)$.) If interval extension is applied to this version of $f(x)$, the result will be

$$F_c(X) = f(c) + Y_1 H_1(Y) + \ldots + Y_n H_n(Y). \tag{6-5-4}$$

In this relation, the subscript c in F_c denotes the centered form. Also note that the interval Y in this case equals $Y = X - c = X - m(X)$. (In fact, this is why they call this the centered form.)

So what's the use of all this rewriting? Well, the resulting interval extension $F_c(X)$ often results in a closer bound on $f(X)$ then the original interval extension $F(X)$. In fact, it can be shown that the excess width $w(E(X))$ now is proportional to $w(X)^2$ instead of $w(X)$. So if one applies refinement to $F_c(X)$, then $w(E_N(X)) \leq Kw(X)/N^2$ for some constant K.

6-5-2 The mean value form

Define $D_i F(X)$ to be the interval extension of the partial derivative $\partial f(x)/\partial x_i$. This term $D_i F(X)$ actually represents an enclosure of the slope $\partial f(x)/\partial x_i$ over the interval X. This enclosure is used when applying the **mean value form** (MVF). In fact,

$$F_{mv}(X) = f(y) + \sum_{i=1}^{n} D_i F(X)(X_i - y_i), \tag{6-5-5}$$

where generally y is chosen to be the mean (midpoint) of the interval, so $y = m(X)$ and $y_i = m(X_i)$. $F_{mv}(X)$ is called the **mean value extension of f on X**. Just like the CF, the MVF also has an excess width $w(E(X))$ of order $w(X)^2$,

6-5-3 The slope form

In the **slope form** (SF), slopes are examined. The **slope** $s(f, x, y)$ between the points $f(x)$ and $f(y)$ is defined such that

$$f(y) - f(x) = s(f, x, y)(y - x). \tag{6-5-6}$$

The idea of this slope s can be extended to the interval domain. The **slope interval** $S(f, X, y)$ is an enclosure of all slopes $s(f, x, y)$ with $x \in X$.

The above definition of slope works well for functions f with only one parameter, but if x is a vector, things get somewhat harder. In this case, the **slope in the direction** x_i is defined such that

$$f(y) - f(x) = s(f, x, y)(y_i - x_i). \tag{6-5-7}$$

The **slope interval in the direction** x_i (denoted as $S_i(f, X, y)$) is then defined as the enclosure of all slopes $s_i(f, x, y)$ with $x \in X$.

Now it is finally possible to discuss the slope form. This form is very similar to the mean value form, except that $D_iF(X)$ is replaced by $S_i(f, X, y)$. So,

$$F_s(X) = f(y) + \sum_{i=1}^{n} S_i(f, X, y)(X_i - y_i), \tag{6-5-8}$$

where again often $y = m(X)$ is chosen. An interesting fact is that, if the same point y is used for both the MVF and the SF, then $S_i(f, X, y) \subseteq D_iF(X)$ and thus $F_s(X) \subseteq F_{mv}(X)$. In other words, the SF will always perform at least as good as the MVF. The big downside to the SF is that it can sometimes be very hard to find good bounds for $S_i(f, X, y)$.

6-5-4 The monotonicity test form

In the **monotonicity test form** (MTF), a close look is given to the differentials $D_iF(X)$. Suppose that $D_1F(X) \geq 0$. In other words, the partial derivative $\partial f/\partial x_1$ is non-negative for all $x \in X$. In that case, $f(x)$ is minimized if $x_1 = \underline{X}_1$. Similarly, it is maximized if $x_1 = \overline{X}_1$. That is great to know, since now it's not necessary anymore to take into account x_1 when trying to find the minimum or maximum of $f(x)$. It is then possible to simply apply (for example) the mean value form to the remaining parameters. (Note that a nearly identical trick can be applied if $D_1F(X) \leq 0$. Then the minimum occurs when $x_1 = \overline{X_1}$.)

The above idea is the main idea behind the monotonicity test form. First define $\mathcal{S}$ as the set of all indices i with $\underline{D_iF(X)} < 0 < \overline{D_iF(X)}$. (So these are all indices for which the monotonicity trick does not work.) Then the MTF is given by

$$F_{mt}(X) = [f(u), f(v)] + \sum_{i \in \mathcal{S}} D_iF(X)(X_i - y_i), \tag{6-5-9}$$

where the elements of the vectors u and v are given by

$$(u_i, v_i) = \begin{cases} (\underline{X}_i, \overline{X}_i) & \text{if } D_iF(X) \geq 0, \\ (\overline{X}_i, \underline{X}_i) & \text{if } D_iF(X) \leq 0, \\ (y_i, y_i) & \text{otherwise.} \end{cases} \tag{6-5-10}$$

So if either $D_iF(X) \geq 0$ or $D_iF(X) \leq 0$, then the monotonicity trick is applied to parameter x_i. Otherwise, x_i is involved in the MVF.

6-5-5 The Skelboe-Moore algorithm

The **Skelboe-Moore algorithm** is an improved version of the refinement procedure. Suppose that one wants to find the minimum $\min_X(f(x))$ of a function $f(x)$ on an interval X, or at least an accurate lower bound $\underline{F(X)}$ of this minimum. The refinement procedure simply divides X in N^n regions $Z^{(i)}$, finds $\overline{F(Z^{(i)})}$ for all regions i and then selects the lowest value of $\underline{F(Z^{(i)})}$ as the lower bound. (Any of the earlier discussed methods can be used to find $\overline{F(Z^{(i)})}$.) The Skelboe-Moore algorithm tries to search more efficiently.

In the Skelboe-Moore algorithm, all intervals Z are kept in a list $\mathcal{L}$. This list is ordered on $\underline{F(Z)}$, so the first element of the list is always the interval Z with the lowest value of $\underline{F(Z)}$. The algorithm is now executed as follows.

1. **Initialize $\mathcal{L}$ as $\{X\}$.** So initially $\mathcal{L}$ contains only X.

2. **Take the first element out of $\mathcal{L}$ and call it Z.** Note that this is the element with the smallest value of $\underline{F(Z)}$ so far.

3. **Check if $F(Z)$ is narrow enough.** That is, check if $w(F(Z)) < \epsilon$, with ϵ the required accuracy. If so, jump to step 6. If not, continue with step 4.

4. **Bisect Z into $Z^{(1)}$ and $Z^{(2)}$.** To bisect Z, first find the dimension i in which Z has the largest width $w(Z_i)$. (So $w(Z_i) = w(Z)$.) Then split up Z along this dimension. (That is, split up Z_i into two equally wide subintervals $Z_i^{(1)}$ and $Z_i^{(2)}$. Keep all other intervals $Z_j = Z_j^{(1)} = Z_j^{(2)}$ (with $j \neq i$) the same, and assemble them all again into two interval vectors $Z^{(1)}$ and $Z^{(2)}$.)

5. **Insert $Z^{(1)}$ and $Z^{(2)}$ into $\mathcal{L}$. Jump back to step 2.** Note that $Z^{(1)}$ and $Z^{(2)}$ should be inserted such that the ordering of $\mathcal{L}$ is maintained. So, when they're 'good' enough to be considered later, they'll automatically drift to the front of the list $\mathcal{L}$.

6. **Algorithm result: an accurate lower bound $\underline{F(X)}$ for $f(x)$ is given by $\underline{F(Z)}$.** Note that it must be the case that $\min_X(f(x)) \in \overline{F(Z)}$. And since $w(F(Z)) < \epsilon$, it must therefore also be true that $\min_X(f(x)) - \underline{F(Z)} < \epsilon$. This implies that the lower bound that has been found has at least an accuracy ϵ.

By the way, it is not always certain that the value x_{min} that minimizes $f(x)$ is in Z. x_{min} might also be in one of the other intervals $Z' \in \mathcal{L}$. Though if $x_{min} \in Z' \neq Z$, then Z' must satisfy $\underline{F(Z')} \leq \overline{F(Z)}$. So if it is necessary to find an interval that bounds x_{min}, then the above algorithm needs to be slightly expanded. It should then, in the end, examine all intervals $Z' \in \mathcal{L}$ with $\underline{F(Z')} \leq \overline{F(Z)}$.

Finally, note that finding a lower bound for a function $f(x)$ is equivalent to finding an upper bound for a function $-f(x)$. So the Skelboe-Moore algorithm can be used to find upper bounds $\overline{F(X)}$ as well. In this way, a bound $F(X)$ of $f(X)$ with any accuracy ϵ can be obtained.

6-6 Linear systems of equations

Solving only one-dimensional functions is a bit boring. In this section linear systems of interval equations will be considered. First the workings of interval matrices will be examined. After that, several methods to solve linear systems of equations will be discussed.

6-6-1 Interval matrices

An **interval matrix** is a matrix whose elements are intervals. These matrices may have various properties.

- If A is an interval matrix and B is a degenerate matrix (with real numbers), and if $B_{ij} \in A_{ij}$ for all i, j, then one may write $B \in A$.
- The **matrix norm** $||A||$ is an extension of the **maximum row sum norm** that is occasionally used for degenerate matrices. So,

$$||A|| = \max_i \sum_j |A_{ij}|. \tag{6-6-1}$$

  In other words, to find the matrix norm, examine all rows. The row with the largest sum of interval norms gives the matrix norm. If B is a degenerate matrix with $B \in A$, then always $||B|| \leq ||A||$.
- The **width** $w(A)$ of an interval matrix A is defined as the maximum of the widths of the elements. So,

$$w(A) = \max_{i,j} w(A_{i,j}). \tag{6-6-2}$$

- The **midpoint** $m(A)$ of an interval matrix is the matrix whose elements are the midpoints of the elements of A. So,

$$(m(A))_{i,j} = m(A_{i,j}). \tag{6-6-3}$$

- The **matrix determinant** $\det(A)$ can be found in the same way as for a degenerate matrix, except this time the result will be an interval.
- Adding, subtracting and multiplying interval matrices is identical to adding, subtracting and multiplying degenerate matrices, except this time interval arithmetics are applied.
- If A and B are interval matrices and C and D are degenerate matrices satisfying $C \in A$ and $D \in B$, then it always holds that $C \odot D \in A \odot B$, where $\odot$ can be any of the above three operations. In case of adding and subtracting, $A \pm B$ forms a tight bound for $C \pm D$. However, in case of multiplying, there will almost always be matrices $E \in AB$ that cannot be the result of any multiplication CD. (This is the result of interval dependency.)

6-6-2 Solving a simple matrix interval equation

Examine the matrix interval equation $Ax = B$ with A an interval matrix and B an interval vector. If A is a 2×2 interval matrix, the equation will look like

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} x = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}. \tag{6-6-4}$$

Finding the set X of all possible solutions x is generally very hard. Also, the solution space often has a shape that is nowhere near a rectangle (or an n-dimensional box). So describing all possible solutions of x with an interval vector X will result in a very rough bound. The question remains, how can such a rough bound be obtained?

One way is to try to solve the system algebraically. For this, first rewrite the set of equations to

$$A_{1,1}x_1 + A_{1,2}x_2 = B_1 \quad \Rightarrow \quad A_{1,1}A_{2,1}x_1 + A_{1,2}A_{2,1}x_2 = A_{2,1}B_1, \tag{6-6-5}$$

$$A_{2,1}x_1 + A_{2,2}x_2 = B_1 \quad \Rightarrow \quad A_{1,1}A_{2,1}x_1 + A_{1,1}A_{2,2}x_2 = A_{1,1}B_2. \tag{6-6-6}$$

Subtracting the first equation from the second will eliminate x_1. (Note that with interval arithmetics, it generally does not hold that $C - C = 0$ for an interval C. But if both intervals C represent the same real number c, then this $C - C = 0$ can be applied.) Solving for x_2 will then give

$$x_2 \in \frac{A_{1,1}B_2 - A_{2,1}B_1}{A_{1,1}A_{2,2} - A_{1,2}A_{2,1}}. \tag{6-6-7}$$

x_1 can be solved for in a similar way. Do note that solutions only occur if $0 \notin A_{1,1}A_{2,2} - A_{1,2}A_{2,1} = \det(A)$. If it does occur that $0 \in \det(A)$, then the matrix interval equation cannot be solved.

6-6-3 The Krawczyk method

Another way to solve a matrix interval equation is the **Krawczyk method**. In this method, both sides of the equation are multipled by a matrix P and rewritten to

$$x = PB + (I - PA)x = PB + Ex, \tag{6-6-8}$$

where E is defined as $E = I - PA$. The goal is to have $||E|| < 1$. A matrix P which often accomplishes this is $P = (m(A))^{-1}$ (or an approximation of it). If $||E|| < 1$, then the above relation can be used to iteratively solve for the solution X. One starts with the interval vector $X^{(0)}$ whose elements are given by

$$X_i^{(0)} = \frac{[-1, 1]||Pb||}{1 - ||E||}. \tag{6-6-9}$$

If a solution x is in $X^{(k)}$, then it must also be in $PB + EX^{(k)}$. (This follows from the requirement $||E|| < 1$.) The solution bound $X^{(k)}$ can therefore be iteratively refined using the nested sequence

$$X^{(k+1)} = \left(PB + EX^{(k)}\right) \cap X^{(k)}. \tag{6-6-10}$$

And, since the sequence is nested, it will converge.

6-6-4 The Hansen-Sengupta method

The **Hansen-Sengupta method** is an interval extension of the original **Gauss-Seidel iteration**. To apply it, again a matrix P is required (with often $P \approx (m(A))^{-1}$) to turn the equation $Ax = B$ into $Gx = C$ with $G = PA$ and $C = PB$. The iteration is now done according to

$$X_i^{(k+1)} = \frac{1}{G_{i,i}} \left(C_i - \sum_{j=1}^{i-1} G_{i,j} X_j^{(k+1)} - \sum_{j=i+1}^{n} G_{i,j} X_j^{(k)} \right) \cap X_i^{(k)}. \tag{6-6-11}$$

The downside of this method is that it does not work if $0 \in G_{i,i}$ for some i. In that case, a division by zero would occur, which results in problems. There are tricks to work around these problems, but these tricks are outside the scope of this summary.

6-7 Nonlinear systems of equations

In the previous chapter, linear systems of equations have been considered. In this chapter, it will be examined how to solve nonlinear systems of equations. First the simple one-dimensional case will be considered. Then two solving methods for the multi-dimensional problem will be looked at.

6-7-1 The univariate Newton method

Consider a one-dimensional (nonlinear) function $f(x)$. Suppose the goal is to find a solution to $f(x) = 0$ on some interval $X^{(0)}$. To do this, it is possible to apply **Newton's method in one dimension**. The **mean value theorem** states that, for every x and y, there is an s between x and y such that

$$f(x) = f(y) + f'(s)(x - y). \tag{6-7-1}$$

So any solution x to $f(x) = 0$ would have to satisfy

$$f(y) + f'(s)(x - y) = 0 \qquad \Leftrightarrow \qquad x = y - \frac{f(y)}{f'(s)} \tag{6-7-2}$$

for any $y \in X^{(0)}$. It is often conveniently chosen that $y = m(X_0)$. Also denote the interval of all possible values of $f'(x)$ with $x \in X$ as $F'(X)$. Then it is known that the solution x must lie in

$$N(X^{(0)}) = m(x) - \frac{f(m(x))}{F'(X^{(0)})}. \tag{6-7-3}$$

This procedure can be recursively refined using nesting, giving

$$X^{(k+1)} = X^{(k)} \cap N(X^{(k)}). \tag{6-7-4}$$

The sequence $\{X_k\}$ will then converge to an interval X^* which contains all solutions $f(x) = 0$ with $x \in X^{(0)}$.

6-7-2 Extended interval arithmetics

A problem occurs with the above algorithm if $0 \in F'(X)$. In normal interval analysis it is not possible to divide by an interval containing zero. As a solution, **extended interval arithmetics** can be used, which do allow this. If $X = [a, b]$, then

$$\frac{1}{X} = \begin{cases} [1/b, +\infty) & \text{if } a = 0 < b, \\ (-\infty, 1/a] \cup [1/b, \infty) & \text{if } a < 0 < b, \\ (-\infty, 1/a] & \text{if } a < 0 = b. \end{cases} \tag{6-7-5}$$

The problem then is that it's possible to wind up with two intervals instead of one. Using an interval hull won't work either, since that would give $(-\infty, \infty)$. The solution is to simply consider each of the two intervals separately. So first Newton's method is applied for one interval, and then it is applied again for the other interval. In this way, the Newton method can find all zeroes of a function $f(x)$ in a given interval X.

6-7-3 The nonlinear Krawczyk method

In the world of interval analysis, of course there aren't only linear interval equations and one-dimensional interval equations. Now, examine the system of nonlinear equations

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0, \\ &\vdots \\ f_n(x_1, \ldots, x_n) &= 0. \end{aligned} \tag{6-7-6}$$

This system is often written in vector form, as $f(x) = 0$. The question is, how can one find solutions $x \in X$ for this system of equations using interval analysis?

The first thing that needs to be done is determine whether there actually is a solution. Luckily there is a trick for that. First, define $f'(x)$ to be the Jacobian matrix with as elements $(f'(x))_{i,j} = \partial f_i(x)/\partial x_j$. The interval extension of this matrix is denoted as $F'(X)$. Also P is defined to be a nonsingular approximation of $f'(m(X))^{-1}$. And finally, y is taken to be any vector $y \in X$. Now define $K(X)$ as

$$K(X) = y - Pf(y) + \big(I - PF'(x)\big)\,(X - y). \tag{6-7-7}$$

It can be shown that if $K(X) \subseteq X$, then there must be a solution $x \in X$ for the system of equations $f(x) = 0$. (The reasoning behind this is just a bit too complicated to explain in this summary.)

The above function $K(X)$ is used in the **Krawczyk method**. (The Krawczyk method for linear systems was already discussed in the previous chapter. Now the method is extended for nonlinear systems.) During every iteration in this method, the matrix $m(F'(X^{(k)}))^{-1}$ is approximated as P. It is then checked if

$$||I - PF'(X^{(k)})|| \leq ||I - P^{(k-1)}F'(X^{(k-1)})||. \tag{6-7-8}$$

If this relation holds, then the approximation P is better than the old approximation $P^{(k-1)}$ and $P^{(k)}$ is set to P. If not, then the old approximation is better and again $P^{(k)} = P^{(k-1)}$ is used. Also, $y^{(k)}$ is defined as $y^{(k)} = m(X^{(k)})$. The iteration is then finished by updating

$$X^{(k+1)} = K(X^{(k)}) \cap X^{(k)}. \tag{6-7-9}$$

The sequence $\{X^{(k)}\}$ will converge to a limit X^* containing the solution x.

6-7-4 The multivariate Newton method

A nonlinear system of equations can also be solved using the Newton method, but to do that, it needs to be extended to multiple dimensions first. This time, consider a vector function $f(x)$. The solution x could be found using

$$x = y - F'(X)^{-1} f(y). \tag{6-7-10}$$

(Note that $F'(X)$ is an interval matrix function now.) However, inverting an integral matrix is not the most convenient course of action. So instead, consider the rightmost term and define it as $V = -F'(X)^{-1} f(y)$. If V is known, then it is also known that the solution x lies in

$$x \in N(X) = y + V. \tag{6-7-11}$$

But how can one find V? Well, the interval V bounds all numbers v satisfying $v = F'(X)^{-1} f(y)$, or, equivalently,

$$F(X)v = f(y). \tag{6-7-12}$$

This is simply a set of linear interval equations, so it is possible to use a linear solving method (like the Gauss-Seidel method or the linear Krawcyk method) to solve for V. Once V is known, $N(x)$ can be found and then the solution interval X can be refined using equation (6-7-4).

6-8 Integrals and interval analysis

Consider a complicated integral that needs to be approximated using interval analysis. How is this done? This section first examines a few simple ways to bound the value of the integral. After that, a look will be given at how integral polynomials can be used to find a more narrow bound.

6-8-1 A few simple ways to bound integrals

Examine a one-dimensional function $f(x)$. This function can be integrated over an interval $X = [a, b]$. This integral is written as either

$$\int_a^b f(x)\,dx, \qquad \int_{[a,b]} f(x)\,dx \quad \text{or} \quad \int_X f(x)\,dx. \tag{6-8-1}$$

If $f(x)$ is continuous, it is easily possible to find bounds on this integral. In fact, a rough bound is given by

$$\int_X f(x)\,dx \in f(X)w(X). \tag{6-8-2}$$

This bound can be refined by splitting up X into N subintervals $X_1, \ldots, X_N$. This gives

$$\int_X f(t)\,dt \in \sum_{i=1}^{N} F(X_i)w(X_i). \tag{6-8-3}$$

If it is also known that the width $w(F(X))$ varies at most linearly with the width $w(X)$ (that is, if $w(F(X)) \leq Lw(X)$ for some constant L), then

$$w\left(\sum_{i=1}^{N} F(X_i)w(X_i)\right) \leq L\sum_{i=1}^{N} w(X_i)^2. \tag{6-8-4}$$

In other words, the width of the integral estimate decreases quadratically with the width of the intervals X_i. So to get a more accurate bound on the integral, one can simply take more (and thus smaller) intervals X_i.

It is also possible to have integrals of interval-valued functions. Such an interval integral is defined as

$$\int_X F(t)\,dt = \left[\int_X \underline{F}(t)\,dt, \int_X \overline{F}(t)\,dt\right]. \tag{6-8-5}$$

The above tricks work the same for this kind of integral.

6-8-2 Integration of interval polynomials

An **interval polynomial** is a polynomial with interval coefficients, like

$$P(t) = A_0 + A_1 t + \ldots + A_q t^q. \tag{6-8-6}$$

When integrating an interval polynomial, one gets

$$\int_{[a,b]} P(t)\,dt = T_0 + T_1 + \ldots + T_q, \tag{6-8-7}$$

where the coefficients are given by either

$$T_i = A_i \frac{(b^{i+1} - a^{i+1})}{i+1} \tag{6-8-8}$$

if i is even or a and b have the same sign, or

$$T_i = \left[\frac{\underline{A}_i b^{i+1} - \overline{A}_i a^{i+1}}{i+1}, \frac{\overline{A}_i b^{i+1} - \underline{A}_i a^{i+1}}{i+1}\right] \tag{6-8-9}$$

if i is odd and a and b have a different sign ($a < 0 < b$).

6-8-3 Interval enclosures

Consider a time-varying parameter $x(t)$. An interval function $X(t)$ that encloses $x(t)$ at every time $t \in T$ is called an **interval enclosure**. If $X(t)$ can also be written as an interval polynomial, then it is called an **interval polynomial enclosure**.

It is possible to obtain an interval polynomial enclosure $X(t)$ of any time-varying parameter $x(t)$ using Taylor series. The Taylor series of $x(t)$, with respect to some point x_0, is given by

$$x(t) = x(t_0) + \sum_{k=0}^{\infty} \left(\frac{1}{k!} \frac{d^k x}{dt^k}(t_0) \right) (t - t_0)^k. \tag{6-8-10}$$

Of course, summing up an infinite amount of terms is computationally not a good idea. If the series is cut off at the K^{th} term, then the result is

$$x(t) = x(t_0) + \sum_{k=0}^{K-1} \left(\frac{1}{k!} \frac{d^k x}{dt^k}(t_0) \right) (t - t_0)^k + r_K, \tag{6-8-11}$$

where r_K is the **remainder term**. From calculus, it is known that an interval enclosure R_K of this remainder r_K is given by

$$r_K \in R_K = \left(\frac{1}{K!} \frac{d^K x}{dt^K}(T) \right) (t - t_0)^K, \tag{6-8-12}$$

where $T_0 = [t_0, t]$. Combining this with the rest of the Taylor polynomial of $x(t)$ will result in an enclosure $X(t)$ of $x(t)$. This gives

$$x(t) \in X(t) = x(t_0) + \sum_{k=0}^{K-1} \left(\frac{1}{k!} \frac{d^k x}{dt^k}(t_0) \right) (t - t_0)^k + \left(\frac{1}{K!} \frac{d^K x}{dt^K}(T_0) \right) (t - t_0)^K. \tag{6-8-13}$$

Note that this is an interval polynomial enclosure of $x(t)$. And since the only interval occurs in the last term, it's a pretty good interval enclosure as well, as long as $w(T_0)$ doesn't become too big.

6-8-4 Finding interval enclosure coefficients

Take a closer look at the coefficients of the interval enclosures. First, for shorter notation, define

$$(x)_0 = x(t_0) \quad \text{and} \quad (x)_k = \frac{1}{k!} \frac{d^x}{dt^k}(t_0). \tag{6-8-14}$$

Finding these coefficients is possible, but may computationally be a bit inefficient, due to all the derivatives. Luckily, there are some tricks that can be used. Consider the time-varying parameters $u(t)$ and $v(t)$, as well as their sum $u + v$. The coefficients of their sum can be found through

$$(u + v)_k = (u)_k + (v)_k. \tag{6-8-15}$$

There are similar tricks for various other combinations/functions of u and v. Some examples include

$$(uv)_k = \sum_{j=0}^{k} (u)_j (v)_{k-j}, \tag{6-8-16}$$

$$\left(\frac{u}{v}\right)_k = \left(\frac{1}{(v)_0}\right)\left((u)_k - \sum_{j=1}^{k} (v)_j \left(\frac{u}{v}\right)_{k-j}\right), \tag{6-8-17}$$

$$(u^a)_k = \left(\frac{1}{u}\right) \sum_{j=0}^{k-1} \left(a - \frac{j(a+1)}{k}\right) (u)_{k-j} (u^a)_j, \tag{6-8-18}$$

$$(e^u)_k = \sum_{j=0}^{k-1} \left(1 - \frac{j}{k}\right) (e^u)_j (u)_{k-j}. \tag{6-8-19}$$

Similar relations exist for many more mathematical functions.

By using the above relations, one can express the coefficients of functions of $(u)_k$ and $(v)_k$ in earlier derived coefficients. This allows for recursively derived coefficients, which is often a more efficient way of finding them than the traditional way involving derivatives.

It is even possible to find the coefficients of a very complicated function, as long as it is written as a sequence of basic operations. Simply recursively walk through all the operations until all coefficients have been found.

6-8-5 Integration of polynomial interval enclosures

Once an interval enclosure $X(t)$ of $x(t)$ for some t_0 has been obtained, then it can be integrated over the interval A. The integral of $X(t)$ over A then provides a bound on the integral of $x(t)$ over A.

But what if the interval is too wide, and a more narrow interval is required? There are then two options. First of all, one can vary K. Higher values of K often (though not always) give more narrow intervals.

A second option is to divide A into N subintervals A_i. One can then evaluate $X(t)$ separately for each subinterval A_i. So,

$$\int_A x(t)\,dt \in \sum_{i=1}^{N} \left(\int_{A_i} X(t)\,dt\right). \tag{6-8-20}$$

Denote A as $A = [a, b]$. If a and b have the same sign, then the right-hand side of the above relation can also be further worked out as

$$\sum_{i=1}^{N} \left(\sum_{k=0}^{K-1} \left(\frac{(x)_k(\underline{A}_i) w(A_i)^{k+1}}{k+1} \right) + \frac{(x)_K(A_i) w(A_i)^{K+1}}{K+1} \right). \tag{6-8-21}$$

Now, by varying either or both of N and K, and then taking the intersection of all the resulting bounds, a narrow bound for the integral of $x(t)$ over A can be obtained.

6-9 Bounding differential equation solutions

The final application of IA to be discussed in this summary is to use intervals in bounding solutions of differential equations. This section briefly discusses how that works for ordinary differential equations. Bounding partial differential equations is however outside of the scope of this summary.

6-9-1 Function and interval operators

A **function operator** p is an operator that can be applied to normal functions like $y(t)$. It may contain derivatives, integrals, etcetera of $y(t)$. An example of a function operator p is

$$p(y)(t) = h(t, y(t)) + \int_0^t f(t, s, y(s))\, ds, \tag{6-9-1}$$

where h and f can be any functions. Note that a function operator has a function as both its input and its output.

Similar to the function operator p is the **interval operator** P. This operator takes interval functions like $Y(t)$ as input and has another interval function $P(Y)$ as output. An interval operator can have various properties. An interval operator P is said to be inclusion isotonic if

$$X \subseteq Y \quad \Rightarrow \quad P(X) \subseteq P(Y). \tag{6-9-2}$$

(Note that, for interval functions $X(t)$ and $Y(t)$, it is said that $X \subseteq Y$ only if $X(t) \subseteq Y(t)$ for every t.) Furthermore, an interval operator P is said to be an **interval majorant** of a function operator p if

$$p(y) \in P(Y) \text{ for every } y \in Y. \tag{6-9-3}$$

(Note that, for a function $y(t)$ and an interval function $Y(t)$, it is said that $y \in Y$ only if $y(t) \in Y(t)$ for every t.)

Now consider the example function operator p of equation (6-9-1). If the interval extensions H and F are inclusion isotonic, then the interval operator

$$P(Y)(t) = H(t, Y(t)) + \int_0^t F(t, s, Y(s))\, ds \tag{6-9-4}$$

is an inclusion isotonic interval majorant of p.

6-9-2 Solving ordinary differential equations

It is often necessary to find solutions $y(t)$ of the equation $y = p(y)$, with p of the form (6-9-1). For example, consider the special case where

$$y(t) = y_0 + \int_0^t f(s, y(s))\, ds. \tag{6-9-5}$$

Taking the derivative of this equation gives $dy/dt = f(t, y(t))$ with $y(0) = y_0$. So this is in fact an **initial value problem**. In a similar way $y = p(y)$ can be rewritten to other well-known ordinary differential equation problems.

Finding the function y that satisfies $y = p(y)$ is often pretty hard. Instead of finding it directly, one can also try to bound it by finding the interval function Y that satisfies $Y = P(Y)$.

Suppose that P is an inclusion isotonic interval majorant of p. Now consider the sequence $Y^{(k+1)} = P(Y^{(k)})$. It can be shown that, if $P(Y^{(0)}) \subseteq Y^{(0)}$, then also $Y^{(k+1)} \subseteq Y^{(k)}$ for every k. So the sequence is nested. Furthermore, the limit

$$Y(t) = \cap_{k=0}^{\infty} Y^{(k)}(t) \tag{6-9-6}$$

will converge to a bound on $y(t)$. In the special case that also

$$\sup_t w(P(X)(t)) \leq c \sup_t w(X(t)) \tag{6-9-7}$$

for all interval functions $X \in Y^{(0)}$ (with $c < 1$ a constant), then it can be shown that $w(Y(t)) = 0$. In other words, $Y(t)$ will become a normal (degenerate) function $y(t)$, equaling the solution to $y = p(y)$.

Part III

Literature research

Chapter 7

On convergence of RL with function approximators

Reinforcement learning with function approximation has been considered often in literature. Many articles have been written on it, or subjects related to it. This chapter will go into depth on these articles. First some recent applications of RL with function approximators are considered. After that, articles that discuss issues in applying RL with function approximators are discussed. Finally, some articles giving proofs of convergence and/or divergence of RL with function approximators are examined.

7-1 Applications of RL with function approximation

This section considers several articles that have set up and applied some version of reinforcement learning combined with function approximation.

7-1-1 The block world problem

The first article to consider is (Irodova & Sloan, 2005), in which RL with function approximation is applied to a version of the block world problem. This is a typical problem with a very large number of discrete states, and solving the problem requires careful planning.

To solve this problem, (Irodova & Sloan, 2005) has used Q-learning, combined with a linear value function and hand-picked features. First a feature vector $\mathbf{f}$ with various useful features f_i is set up. The Q-function is then described as $Q(\theta, s, a) = \theta^T \mathbf{f}$. The update law for the parameter vector θ is

$$\theta \leftarrow \theta + \alpha \left(r + \gamma \max_{a'} Q(\theta, s', a') - Q(\theta, s, a) \right) \frac{dQ(\theta, s, a)}{d\theta}. \tag{7-1-1}$$

Though this research shows the promise of function approximators combined with reinforcement learning, the applicability of (Irodova & Sloan, 2005) for the current project remains

limited. The reason for this is that the problems that are considered in the current project do not have discrete states. Also, the system dynamics aren't necessarily linear – in fact, they are often nonlinear – thus questioning the usefulness of using a linear value function.

7-1-2 Least-squares policy iteration

The article (Lagoudakis, Parr, & Bartlett, 2003) discusses and applies the **Least-Squares Policy Iteration** (LSPI) algorithm. This algorithm uses a linear function approximator, which approximates the Q-function. It does not use a function approximator for the policy. Instead, it directly searches for the action a that maximizes the Q-function.

The LSPI uses linear function approximators, which does not exactly correspond with the subject of the current project. Nevertheless, the algorithm is interesting to analyse. (Lagoudakis et al., 2003) contains a theorem that the LSPI algorithm is stable and always converges to a region near the optimal performance. (So not a local optimum but the global optimum.) However, no proof for this theorem is given, so it cannot be verified.

(Lagoudakis et al., 2003) also discusses an experiment of a cart and pendulum system, very similar to the one considered in the current project. However, there are some issues with this experiment. First of all, it is said that the displacements given to the pendulum in the experiment are very small. Second, it is noted that, *'With 1000 training episodes even the worst policy could balance the pendulum for at least half the time.'* In other words, there were policies that regularly failed to balance the pendulum. This statement makes the validity of the earlier theorem very doubtful. Nevertheless, the article does show that combining least-squares techniques with linear function approximators can be a powerful way to solve certain RL problems.

7-1-3 Cerebella model articulation controllers

In (Zheng, Luo, & Lv, 2006), a double inverted pendulum is controlled using reinforcement learning. As function approximator, a **Cerebella Model Articulation Controller** (CMAC) is used. This type of controller puts several 'grids' over the state space. Every state thus falls in a cell of each grid. These cells are subsequently activated. Every cell also has a weight. The output of the network then becomes the weighted sum of all activated cells.

What is novel, is that the controller in (Zheng et al., 2006) does not use one CMAC controller but two. One controller has very generalized learning, while the other controller has more specialized learning. If a state hasn't been visited much, then (slightly simplified) the generalized network is used, while if it has, the specialized network is used. This is an interesting solution to the generalization/specialization dilemma.

The two CMAC controllers do eventually manage to control the double inverted pendulum. However, they do require a significantly long training time for this. (Almost 500.000 trials are required.) Furthermore, though a CMAC controller can be applied to continuous problems, its working principle resembles more that of a discrete controller than that of a continuous controller. A CMAC controller is thus not suitable for problems like the one considered in the current project, which has a highly continuous nature.

7-1-4 A SARSA controller

Reinforcement learning is also suitable at playing games. This has of course been proven by the famous TD-gammon controller. One other instance where this is shown is (Beitelspacher, Fager, Henriques, & Mcgovern, 2006). Here the SARSA method is used to play a variant of the Spacewar game. Though the SARSA controller does not manage to come close to the optimal performance (for as much as an 'optimal performance' exists in a computer game with a lot of randomness and complex opponents), it does achieve adequate performance. It is also found that the way in which the state is fed to the controller significantly influences the learning performance of the controller.

7-1-5 Adaptive critic designs

One article which considers several variations of RL is (Prokhorov & Wunsch, 1997). This article looks at various RL architectures – how to train the actor (policy) and critic (value function). For example, in **Heuristic Dynamic Programming** (HDP), it is attempted to minimize the magnitude of

$$V(s_k) - \gamma V(s_{k+1}) - r, \tag{7-1-2}$$

while in **Dual Heuristic Programming** (DHP) one tries to minimize the derivative

$$\frac{\partial V(s_k)}{\partial s} - \gamma \frac{\partial V(s_{k+1})}{\partial s} - \frac{\partial r}{\partial s}. \tag{7-1-3}$$

Finally, in **Global Dual Heuristic Programming** (DHP), one tries to minimize the above two quantities simultaneously, through two different learning rates. This method is rather complicated, especially because finding the derivatives will be hard, but (Prokhorov & Wunsch, 1997) claims (without showing actual experiment results) a theoretical increase in learning performance.

7-2 Issues in reinforcement learning with function approximation

Literature has several articles and papers on issues in reinforcement learning as well. This section takes a look at a few important ones.

7-2-1 Value overestimation

(Thrun & Schwartz, 1993) offers a research on issues in reinforcement learning and function approximators, very similar to the goal of this report. However, the authors' approach is very different. They assume that the number of applicable actions is finite – in fact, strongly bounded – and focus on Watkin's Q-learning algorithm, combined with function approximators. They note that, when estimating the value of a state using Q, there will always be an error ϵ. They assume this error is normally distributed. Subsequentially, during the update rule of Q-learning (relation (3-3-5)), it will be necessary to find

$$\max_a Q(s, a). \tag{7-2-1}$$

Because there is an error in Q, this maximization will generally result in a too high value: an overestimate of the value. The result will be an incorrect value function and thus possibly also an incorrect policy.

(Thrun & Schwartz, 1993) offers various ways to reduce the effects of this overestimation. First it provides bounds on γ, such that the effects of overestimation will not significantly influence the policy. Later, it also suggests to add pseudo-costs to the update relation, to counteract for value overestimation.

Though it is a good article on a very similar topic, the value of (Thrun & Schwartz, 1993) for the current project remains limited. This is mainly because in (Thrun & Schwartz, 1993) it has been assumed that the number of actions is limited, while in the current project the action is continuous, and there are thus infinitely many actions possible. This prevents the theory that is offered from being applied.

7-2-2 Issues in function approximation

Another article which describes issues in using function approximation with reinforcement learning is (Motta Salles Barreto & Anderson, 2008). It mentions various problems that can occur.

- In function approximation, when the value $V(s)$ of some state s is updated, the value $V(s')$ of nearby states s' also gets updated. Although for most learning algorithms it is possible to ensure that $V(s)$ becomes more accurate, the same cannot always be said of $V(s')$.
- Sometimes it's not possible for the function approximator to sufficiently approximate the actual value function V. And even if it is possible, it's not always possible to represent intermediate versions of V.
- Finally, even if a good approximation of the value function has been obtained, the fact that it is an approximation may still cause the resulting policy π to perform poorly.

(Motta Salles Barreto & Anderson, 2008) goes on to describe a way in which some of the issues mentioned in both (Thrun & Schwartz, 1993) and (Motta Salles Barreto & Anderson, 2008) can be circumvened. The article describes a Restricted Gradient-Descent (RGD) algorithm. The value function that is applied uses radial basis functions. The 'restricted' part of the RGD algorithm implies that the widths σ of these RBFs may not increase. They can only decrease. Furthermore, the center $\mathbf{c}$ of the RBFs may only move towards states s that have actually been visited. This confines the centers $\mathbf{c}$ to the convex hull of the visited state space. Since both σ and $\mathbf{c}$ are bounded, no divergence can occur during learning.

Furthermore, (Motta Salles Barreto & Anderson, 2008) applies a very interesting addition to the neural networks it uses. Instead of using a fixed number of hidden neurons, the number of hidden neurons increases as the neural network learns. When the algorithm detects that additional 'resolution' is required (i.e., none of the neurons is sufficiently activated), an additional neuron is added at the current state s.

7-2-3 Divergence due to error amplification

Another article that examines convergence issues in reinforcement learning is (Boyan & Moore, 1995). This article distinguishes four kinds of results of the learning algorithm. **Good convergence** occurs when the value function approximator approximates the real value function well, thus resulting in the right policy. **Lucky convergence** takes place when the value function approximator does not converge to the actual value function, but still the right policy is derived. There is **bad convergence** when wrong (suboptimal) value functions and policies are derived. Finally, there is **divergence** when the value function diverges.

(Boyan & Moore, 1995) argues that the most important reason for divergence is the amplification of errors due to the dynamic programming step. The function approximators are updated through the quantity $V(s_k)-\gamma V(s_{k+1})-r$, but when $V(s_{k+1})$ is inaccurate, this update can be wrong, potentially resulting in divergence. To prevent this, (Boyan & Moore, 1995) proposes an algorithm called **Grow-support**, which doesn't use TD updates but instead considers the entire 'rollout' of an experiment and uses that to update the value function, just like in Monte Carlo state-value prediction. This is guaranteed to prevent divergence, and it often also results in convergence, though convergence is not guaranteed.

7-3 Convergence proofs

Many attempts have been made in literature to prove the convergence of a reinforcement learning controller with function approximation. No one has succeeded in general, nor come close to it. However, for specific cases convergence proofs have been found.

7-3-1 Linear quadratic regulation

One of the first articles with a convergence proof is (Bradtke, 1993), which examines linear systems with **Linear Quadratic Regulators** (LQRs). It describes a policy iteration algorithm for LQR problems that is proven to converge to the optimal policy. In contrast to standard methods of policy iteration, it does not require a system model. It is a good start into the world of convergence proofs. However, since it is only applicable to systems with linear dynamics, it is not applicable to the current project.

7-3-2 Finite state and action spaces

(Gordon, 2001) considers the SARSA(0) and V(0) algorithms. (The latter was used in the well-known TD-Gammon program.) The article doesn't show that these algorithms converge to a set point, but it does prove that they converge to a region. To be more precise, the algorithms are proven to be stable in the sense that there exist bounded regions which, with probability 1, they eventually enter and never leave. This does not imply that the algorithm converges to a point though, and for some MDPs this occasionally also doesn't occur. It must be noted that in the article, no mention is made what kind of function approximator is used. However, it is assumed that the amount of states and actions are both finite. Thus, these theories are not applicable to the case in which there are continuous states and/or actions, like in the current project.

7-3-3 Convergence of linear function approximators

Another early proof that certain cases of reinforcement learning with function approximators converge is given in (Tsitsiklis & van Roy, 1997). This article has established the convergence of on-line temporal-difference learning with linear function approximators when applied to irreducible aperiodic Markov chains. At the end of the article, it is also shown that convergence is not necessarily the case when nonlinear function approximators are used. An example of an RL agent with a special type of nonlinear function approximator is given that has divergent learning on a specific problem. As such, the article shows that not all RL agents with nonlinear function approximators converge. However, it also poses the question whether this would hold for specific nonlinear function approximators, like NNs, or whether things are different then.

One article which has some very powerful results on the convergence of linear function approximators is (Merke & Schoknecht, 2004). It considers synchronous reinforcement learning with linear function approximation and transforms this process into inhomogeneous matrix iterations. It then provides necessary and sufficient conditions for various kinds of convergence. Since these conditions are both necessary and sufficient, it is also possible to prove divergence, which is a very powerful result. Though this theory evidently is not applicable to nonlinear function approximators, it is definitely a powerful result.

(Melo, Meyn, & Ribeiro, 2008) considers Q-learning combined with linear function approximation. It proves that, given certain conditions, this algorithm converges with probability 1. Its method is very similar to (Merke & Schoknecht, 2004), with as most notable difference that (Melo et al., 2008) uses an ordinary differential equation $\dot{\theta}(t) = A_\theta\theta(t) + b$ to show convergence, whereas (Merke & Schoknecht, 2004) uses a discrete version $\theta(k+1) = A\theta(k) + b$. Nevertheless, the outcome is the same. Given certain conditions, Q-learning with linear function approximation converges.

7-3-4 Expansion to nonlinear function approximators

So far linear function approximators have been considered. Convergence proofs for RL with nonlinear function approximators are very rare. One example of an article that does discuss it is (Sutton, McAllester, Singh, & Mansour, 1999), in which it is shown for the first time that a version of policy iteration with arbitrary differentiable function approximation is convergent to a locally optimal policy. In this article, two function approximators are used: a policy π_θ (with parameter vector θ) and a value function f_w (with parameter vector w), the latter of which serves as an approximation of $Q(s,a) - V(s)$.

In the algorithm, a form of policy iteration is applied. This iteration consists of two steps. First, in the policy evaluation step, w is varied until f_w reaches a local optimum, thus approximating $Q^\pi(s,a) - V^\pi(s)$ as well as possible. Second, in the policy improvement step, θ is adjusted in the direction of the negative gradient $\partial\rho/\partial\theta$, with ρ the expected total reward. It is then shown that this iteration (subject to various conditions) is ensured to converge to a local optimum.

Though the article provides a very powerful proof, it does have a few downsides.

- The proof given in the paper is incomplete. It uses theorems without showing all conditions for these theorems are met. Nevertheless, this problem can be fixed by extending the proof.
- The given algorithm has little practical applicability so far; the reason being that the policy evaluation step is likely to require an infinite number of iterations to converge. Of course, one can terminate the policy evaluation sooner, giving a nearly identical performance, but then the proof given doesn't hold anymore and convergence is not guaranteed.
- The algorithm ensures convergence to a local optimum, and although a convergence guarantee certainly is valuable, convergence to a local optimum that does not equal the global optimum can still be unsatisfactory in practical applications.

One possible improvement to the algorithm described in (Sutton et al., 1999) is thus to change the policy iteration algorithm into a value iteration algorithm, and see if a similar proof can be found. Another option is to combine the given algorithm with a variation of the Skelboe-Moore algorithm of interval analysis (as described in subsection 6-5-5) to try and find a way to ensure convergence to the global optimum.

An article that continues on the work of (Sutton et al., 1999) is (Bhatnagar, Sutton, Ghavamzadeh, & Lee, 2009). It offers four actor-critic algorithms with linear function approximators and provides theoretical convergence proofs to local minima for them. It is the first article that provides a convergence proof of an algorithm that applies bootstrapping both for the actor (policy) and the critic (value function). (In **bootstrapping** methods, estimates are updated based on other estimates. The TD-method is an example of a bootstrapping method.)

Again, the applicability of the article to the current project remains limited. First of all, it applies linear function approximators, while these are likely to be insufficient for the type of nonlinear problems that needs to be solved in the current project. Furthermore, though the convergence proofs of the provided algorithms in the article are very impressive, the downside is that everywhere in the text references are made to so many theorems and claims from other articles that it is nearly impossible to verify the proofs. Finally, none of the algorithms have been tested in an actual experiment. Their effectiveness thus remains very doubtful.

7-4 Summary of issues and potential solutions

This section summarizes the previous sections. What proofs are around? What are the present issues? And how can these issues be circumvented?

It is hard to adequately summarize what has been accomplished, since there are so many different ways of setting up an RL controller with function approximators. What is certain, is that no one has yet proved convergence for a practical application of RL with neural network function approximators. Nor has anyone tried to do so using interval analysis.

Something that has been proven is the convergence of many types of RL controllers with linear function approximators. These types of function approximators are relatively well-understood and it is also relatively easy to derive powerful equations with them. However, the current

project focusses on nonlinear function approximators, so these results are not directly useful for this report.

When it comes to RL with nonlinear function approximators, only one convergence proof has been found. However, this was for an algorithm that is not yet practically applicable. For the more practically applicable algorithms, several issues arise. They will be examined one by one, including potential solutions for them.

- **Problem:** When applying Q-learning with function approximators, then the introduced approximation errors, combined with the maximization step, may result in an overestimation of the Q values.
  Possible solutions: It is possible to introduce pseudo-costs, to use function approximators that only make underestimates and no overestimates, or simply to change the algorithm and get rid of the maximization step. If for example an actor is derived using gradient descent method, then no maximization of the Q function is required, and the problem disappears.

- **Problem:** When the value of $V(s)$ is being adjusted, the value of $V(s')$ is also adjusted, where s' is close to s. (This is due to the generalization that takes place.) Though the value of $V(s)$ generally becomes more accurate, things may be different for $V(s')$.
  Possible solutions: This problem is hard to solve as the generalization is the exact reason why function approximators are used. One possible solution would be to use a function approximator with which it is possible to tune the 'range' of adaptation. If adjustments are made in such a way that only states s' very close to s are affected by an update of s, then this problem will be solved. The immediate downside will of course be a reduced generalization ability and thus a lower learning performance. To circumvent this, one could try lowering the range of adaptation as the algorithm learns. However, this solution is still not optimal.

- **Problem:** Sometimes the function approximator cannot sufficiently approximate the value function, or intermediate stages of it. This may result in learning the wrong value function. Alternatively, it may result in a decent approximation of the value function, which however still results in an incorrect policy.
  Possible solutions: This problem can be solved by using the right function approximator. For example, if an NN is given enough hidden nodes, it can approximate any function. The number of hidden nodes of the network can be increased. This can be done prior to learning, but what would be even more promising is to do this on the fly. By doing this, any function can be approximated as accurately as required. Given enough learning time, the problem should eventually stop to occur.

- **Problem:** Initially, the TD error is based on an inaccurate value function. This incorrect TD error may cause divergence in TD methods.
  Possible solutions: It is known that TD methods use biased estimates of the value function. Using Monte Carlo methods instead (i.e., simply executing an entire episode and then use the final result to update the value function) will result in unbiased estimates of the value function. Furthermore, while the TD error can diverge, the value estimates resulting from Monte Carlo methods are always bounded, so no divergence can occur. A possible solution would thus be to apply Monte Carlo methods until the value

function is sufficiently accurate to switch to TD methods. This switch is warranted by the fact that TD methods generally still adjust faster than Monte Carlo methods.

- **Problem:** Many learning methods are based on gradient descent methods. These methods have as inherent disadvantage that they can converge to a local optimum instead of a global optimum.
  Possible solutions: This is probably the hardest problem to solve. To do so, one needs to keep track of all possible sets of weights that may lead to the global solution. One possible idea to accomplish this is to apply a version of the Skelboe-Moore algorithm to make sure the global optimum is found. Alternatively, there are statistical approaches that may achieve the same, though more of these will be seen in the next chapter. It is not clear yet in which exact way these methods should be applied, but it is clear that they should keep track of the set of possible policies in such a way that they do not miss the global optimum.

It can be concluded that there are still many issues which may cause RL controllers with function approximators to diverge or converge incorrectly. However, ideas to solve all these issues are present. Further research should show whether these possible solutions actually work, and whether they work well enough to ensure convergence of nonlinear function approximators to the global optimum.

Chapter 8

On further relevant ideas

During the literature research, several articles have been found that do not directly concern reinforcement learning with function approximation. However, they do provide interesting tools or inspiritations to improve reinforcement learning algorithms with. First, some possible improvements on neural network learning will be considered. Second, some inspirations by nature will be examined. Finally, a look will be given to any other potential improvements that can be made to RL controllers.

8-1 On neural networks

The function approximator that is closely examined in the current project is the neural network. This neural network may have some potential improvements, which will be considered in this section.

8-1-1 Interval analysis applied to neural networks

In (de Weerdt et al., 2009), methods are explained with which the neural network output can be optimized using interval analysis. Given a neural network, this article discusses methods to maximize (or equivalently minimize) the NN output. Though there are some drawbacks to this method, like a high computational load and dependency effects, it does guarantee that all solutions are found to any degree of accuracy with guaranteed bounds.

Whether the methods discussed in (de Weerdt et al., 2009) will be useful for the current project is still unsure. One might initially think they will not. After all, this project focusses on changing the actual neural networks, and not on finding the optimal output for a fixed neural network. However, consider the hypothetical case where the value function V and the identifier I both converge to the correct values, but the policy π does not. Then the way in which the policy learns should be altered. It is possible, using the methods discussed in (de Weerdt et al., 2009), to exactly find the optimal action a for a state s, given the value

function V and the identifier I. This optimal action can subsequently be used to train the policy π, after which it will converge to the correct values as well. Whether this situation arises still remains to be seen. If it does, this article will be kept in mind.

8-1-2 Statistical approaches to neural networks

A problem of neural networks and its backpropagation algorithm is that it often converges to a local (and not a global) minimum. This especially poses problems for self-learning controllers, which might converge to the wrong behavior. An alternative to the backpropagation method that might solve this issue is offered in (Levin, Tishby, & Solla, 1989), which discusses a statistical approach to learning and generalization in NNs.

In general, the idea of an NN is to approximate a function F by an approximated function F_w. The main idea of (Levin et al., 1989) is to examine the probability $p(y|x, w)$ that y is outputted given the input x and weight set w. The goal now is to find the weights w that maximize $p(y_i|x_i, w)$ for every test sample i. Or, to be more specific, the goal is to maximize the probability $P(X, Y|w)$ that the entire test set X, Y occurred. This probability is given by

$$P(X, Y|w) = \prod_{i=1}^{m} P_F(x_i) \prod_{i=1}^{m} p(y_i|x_i, w). \tag{8-1-1}$$

This relation should be maximized for the weight w. This is however pretty hard to do. So instead, using Bayes formula, the relation is inverted to

$$\rho(w) = P(w|X, Y) = \frac{\rho^0(w) \prod_{i=1}^{m} p(y_i|x_i, w)}{\int_W \rho^0(w) \prod_{i=1}^{m} p(y_i|x_i, w)}. \tag{8-1-2}$$

Here, the quantity $\rho(w)$ describes the distribution of the weights, and $\rho^0(w)$ describes the initial (prior) distribution.

Though it is not directly evident how the theory described in (Levin et al., 1989) can be applied to the current project, the idea of statistically analyzing neural networks does seem promising enough to keep in mind. The main advantage is that this theory does not have a fully defined set of NN weights, but instead continually examines the entire distribution of possible weights. Though computationally this will be very complicated, especially due to the integral over the weight space W, it may be the main part of an idea that would ultimately find the global optimum of NN training.

A different approach, that also uses statistics but in a completely different way, is posed in (Agostini & Celaya Llover, 2010). This article describes the application of a **Gaussian Mixture Model** to reinforcement learning. A GMM is a weighted sum of multivariate Gaussian Probability Density Functions (PDFs), and is used to represent general probability density distributions in multi-dimensional spaces. In the article, a GMM is used to describe the PDF of the joint space of the state s, the action a and the corresponding Q-value.

As the algorithm learns, it updates the GMM. Gaussian PDFs may be added and the strengths, mean vectors and covariance matrices of the existing PDFs are updated. A downside of this algorithm is that regions that are rarely visited are forgotten, but an adjustment is described that solves this problem. In the end, (Agostini & Celaya Llover, 2010) tests the

performance of the algorithm and finds that it is very well capable of learning to swing up a pendulum.

The theory discussed in (Agostini & Celaya Llover, 2010) is very promising, and might very well be suitable to apply to the current project. The only downside of the article is that it avoids the use of an actor, and subsequentially faces the problem of maximizing $Q(s, a)$ manually. It does this by simply trying out the value of $Q(s, a)$ for a finite number of actions a. This may very well result in sub-optimal actions, though it may also be possible to create a method to work around this problem. Besides this issue, the use of GMMs is promising enough to deserve a closer examination in later stages of the current project.

8-2 Inspirations from nature

Reinforcement learning is a technique that is derived from nature. It is very similar to the way a human, or almost any animal for that matter, learns. This means that, when looking for ways to improve the reinforcement learning algorithm, one can also look at and be inspired by nature. Of course, this has already been done many times in literature.

8-2-1 Inspirations from scientific articles

(Daw & Frank, 2009) describes a brief literature survey in which various research groups have looked closely at animals, and even have done experiments with animals, to derive more about the way they learn.

One example of such a research project is described in (Huys & Dayan, 2009). This article describes **learned helplessness**. First an agent (e.g., a mouse) is subjected to punishments (e.g., electric shocks). The agent may have control over when to stop these punishments (e.g., by pushing a button), or he may not. (Huys & Dayan, 2009) found that in the latter case, the agent is subject to learned helplessness. The next time that the agent is subjected to punishments, the agent will not even try to make it stop, even if it has the option to.

This interesting concept can be translated to RL. It shows that, in nature, the training of the value function and the policy go hand in hand. If, during initial training, the agent can stop his punishments, he will learn a good policy: try to find a way to stop the punishments. On the other hand, if he cannot stop his punishments, then this will also reflect in the policy. The policy will simply be 'take no action in case of punishments, since it is of no use anyway.' It is thus important to directly train the agent on a scenario that is as representative as possible.

8-2-2 Inspirations from popular literature

Popular literature also offers various books on how humans learn. Since reinforcement learning is so similar to how humans learns, using the theories in these books can provide a lot of inspiration on how to improve the RL algorithm.

Intrinsic motivation

In (Pink, 2010), it is described how humans have an intrinsic drive to achieve mastery: to learn something useful and to learn it well. People often do things not because they get a reward, but because they like to get better at something. In reinforcement learning, such a drive is not present. Only rewards matter. This gives rise to the idea of implementing intrinsic motivation into reinforcement learning. That is, unfamiliar actions that lead to the learning of new behavior should be rewarded as well. How exactly this should be implemented in detail is something that still requires further thought though.

Short-term versus long-term thinking

Another interesting perspective on learning is offered in (Swaab, 2010), which discusses the working principles of an actual human brain from a brain surgeon's point of view. It is told that the prefrontal cortex of human children is not fully developed yet. This part of the brain is involved in planning and long-term thinking. Only near adolescence will the prefrontal cortex fully mature. It is only then that people will be fully capable of thinking long-term. It is argued that this late development of the prefrontal cortex improves learning at early ages. People first need to learn what is important to do in the short term to be able to think about the long-term effects of their actions.

How can this idea be implemented in the reinforcement learning algorithm? The parameter that determines long-term thinking is the discount factor γ. A high value of γ ensures long-term thinking, while a low value causes short-term thinking. So this gives rise to the idea of starting off with a low value of γ, and increasing it as the controller learns.

But is it possible to do this? Yes, but not without some minor adaptations, as otherwise this would result in conflicts with the theory presented in section 5-3. This section told that the value function should approximate the total (weighted) reward

$$R_k = \sum_{i=0}^{\infty} \left(\frac{\gamma^{(t_{k+i+1}-t_k)} - \gamma^{(t_{k+i}-t_k)}}{\ln \gamma} \right) r_{k+i+1}. \tag{8-2-1}$$

Now, if γ is increased, the weights in this relation will increase as well, and so will the value of R_k. Since the NN value function approximation is not directly adapted, this would result in an incorrect value function approximation. To solve this, the value function approximator should not approximate the actual total reward R_k (since this varies with γ) but the weighted average reward $\bar{R}_k$ (which does not directly vary with γ). It is given by dividing R_k by the sum of the weights. So,

$$\bar{R}_k = \frac{\sum_{i=0}^{\infty} \left(\frac{\gamma^{(t_{k+i+1}-t_k)} - \gamma^{(t_{k+i}-t_k)}}{\ln \gamma} \right) r_{k+i+1}}{\sum_{i=0}^{\infty} \left(\frac{\gamma^{(t_{k+i+1}-t_k)} - \gamma^{(t_{k+i}-t_k)}}{\ln \gamma} \right)} = \sum_{i=0}^{\infty} \left(\gamma^{(t_{k+i}-t_k)} - \gamma^{(t_{k+i+1}-t_k)} \right) r_{k+i+1}, \tag{8-2-2}$$

or, to be really short, $\bar{R}_k = -\ln(\gamma) R_k$. Implementing this idea and having γ increase accordingly should speed up the learning process.

Muscle learning

Another interesting insight comes from the field of muscle learning. When learning a new task, it's not only the brain that learns. It's also the muscles. And the way muscles do this is interesting to look at.

Imagine you're learning a new task that requires precision, like throwing a dart. Initially, your muscles cannot provide this precision. Although in theory you know exactly how to execute the task, your muscles still cannot throw the dart in the right place all the time. Sometimes you have a hit, but sometimes it's too high or too low. There's too much variance in the execution. The theory says that this variance is present so you learn the right behavior. It's the 'exploration' required by the muscles. As you practice more, this variance decreases though, until you can throw the dart almost perfectly almost all the time.

This theory of muscle learning can also be used to get rid of some flaws in common solutions to the exploration/exploitation dilemma. For example, when applying the ϵ-greedy policy, a fully random action is regularly taken. But for some systems one (badly chosen) random action might mess up the process it's been working towards for several time steps. Instead, the solution that nature suggests is to simply follow the policy $\pi(s)$ but, on top of this, to add a noise with variance $\delta(s)$. This variance $\delta(s)$ depends on how familiar (i.e., well-trained) the agent is with the state s. Initially $\delta(s)$ is big, but as s (or similar states) are visited often, its value decreases. This offers an alternate solution to the exploration/exploitation dilemma without suffering the drawbacks of the ϵ-greedy policy.

Learning by example

Another way in which brains in nature work is by imitation and example. Babies mimic their parents. They try to do the same actions as their parents, because they implicitly assume that whatever their parents are doing must work well, so is worth a try. This idea can also be implemented in the RL algorithm. It doesn't suggest that RL controllers should have parents. Instead, they should have something to mimic. Something should show or tell them which actions to try.

Imagine a control problem which already has a conventional (non-adapting) controller that has adequate performance. However, the goal is to train a reinforcement learning controller for this problem. In this case, the older conventional controller can instruct the RL controller on how to control the system – that is, which actions to try. This will cause the RL controller to get adequate performance much faster. Once this is obtained, the conventional controller is discarded and the RL controller keeps on learning on its own. (That is, the child leaves the nest.) This trick could speed up RL controller learning. It does also steer the way in which it learns, which can sometimes be an advantage, yet sometimes also a disadvantage.

Making associations

A final book that provides interesting insights is (Hawkins & Blakeslee, 2004), which discusses the working principles of the brain from a computer programmer's background. It describes how the brain makes associations. When two certain types of 'input' (say, the image of a fly on your arm, and a tickling sensation) always occur together, an association is made between

the two. This association is coupled with a concept: 'there is a fly on my arm.' If some time later only one of these inputs arise (say, a tickling sensation on your arm), the other input is immediately expected (seeing a fly on your arm). This method of 'predicting inputs' is one advantage. However, when the inputs don't match (that is, you don't see a fly on your arm), the result will be surprise. This surprise is the cue that tells the brain to adjust the associations it has.

The process of forming associations and concepts is the key to real intelligence. If it is achieved, the possibilities are legion. For example, it could be implemented in an aircraft. When subsequently one of the aircraft sensors gives a wrong signal, the intelligent controller would notice an association that doesn't hold up. It is 'surprised.' It can firstly inform the pilots of this surprise, and secondly discard the faulty controller data and still control the aircraft appropriately by knowing what kind of sensor data should have been present instead.

Sadly, at the moment no methods are known to successfully implement the whole associating process just like it works in an actual brain. Some methods (like (Agostini & Celaya Llover, 2010), discussed earlier) seem promising to actually achieving it, but they would require much further research.

8-3 Other possibilities of potential future examination

After a reinforcement learning controller has been developed, including adequate neural networks and a good nature-inspired learning algorithm, there are still possibilities for improvement. This section discusses some of them.

8-3-1 Multi-agent systems

Imagine a world in which reinforcement learning controllers with function approximators become common. The result will then be a **multi-agent system** (MAS). A good and broad introduction to MASs is given in (Stone & Veloso, 2000). One thing that becomes very clear from this article is that the agents have the important need to interact. Only if they know each other's behavior and adjust to it, can they either successfully fulfill tasks together if they have common goals, or manage to give each other a hard time if they have conflicting goals.

The question now arises: how can this agent-to-agent interaction be implemented into reinforcement learning controllers? Previously in chapter 5, it has been described how an agent can use an identifier I to learn the behavior of its environment. In a similar way will it be possible to implement an **agent identifier** I_a for every other agent with whom is interacted. In this way, every agent learns the way the other agents behave. As time progresses, and this behavior is learnt, some sort of 'trust' can be established between agents.

One might initially think that this just results in a lot of redundant processing time. After all, if an agent knows exactly what another agent is doing, why would that other agent be needed? This train of thought is incorrect though. The agent identifier I_a only learns the input-output behavior of every other agent. All internal processes, like agent policies and such, do not have to be considered here.

8-3-2 Including rare events

One of the reasons why self-learning controllers can be promising in the real world, is that they can adapt to unexpected circumstances. This can be done in two ways. The first is that the self-learning controller learns on the fly how to deal with an unexpected event. The other is that the controller has already been trained how to deal with an unexpected event earlier.

One downside of the latter method is that it is very hard to train agents sufficiently for events that hardly ever occur. An article that describes how to solve this problem is (Frank, Mannor, & Precup, 2008). With the REASA algorithm that has been developed, it is possible to relatively quickly train RL agents on so-called **rare events** (events that have a very low probability ε of occurring but do significantly influence the value function). The main idea is to use a simulator, in which the probability $\hat{\varepsilon}$ with which these rare events occur can be set manually. Subsequentially, the RL learning algorithm is adjusted to take the altered state transition function into account. The algorithm also provides a way to optimally set the simulated rare event probability $\hat{\varepsilon}$.

8-3-3 Varying the function approximator

In this report, only neural networks are considered, each with only one given layout. Of course the type of function approximator that is used matters for the RL controller performance. Some function approximators will perform better than others, and even for a given function approximator (say, a neural network) the specific layout that is used can have a tremendous effect.

One way in which at least the latter can be improved is by using the NEAT+Q algorithm, proposed in (Whiteson & Stone, 2006). This article combines the **Neuro-Evolution of Augmenting Topologies** (NEAT) technique with Q-learning. NEAT is an evolutionary algorithm in which the structure of function approximators can be varied. For example, for a neural network, the NEAT algorithm can add nodes, add links, etcetera. When combined with Q-learning, the evolutionary algorithm checks various neural networks to see which has the best learning abilities. The best ones are then reproduced into the next generation.

The result of the NEAT+Q algorithm is an improved performance. It generates NN architectures that actually work for the problem at hand. A significant downside is that the algorithm uses a big number of neural networks in every generation. Its run-time is thus relatively high. Furthermore, it gives a lot of added complexity. As such, this technique seems (at least for now) not the most suitable technique to apply in the current project. However, if it later on turns out that the neural network set-up strongly influences the RL controller performance, then the NEAT+Q technique might still be considered.

8-3-4 Relational reinforcement learning

The world of artificial intelligence is very broad. Though reinforcement learning is a very common technique, also **relational learning** (also known as **inductive logic programming**) is very well-known. In this technique, logical relations are used, with which a controller can subsequently plan how to solve a problem.

Attempts have been made to combine these two techniques, resulting in **Relational Reinforcement Learning** (RRL). (Džeroski, De Raedt, & Driessens, 2001) gives a basic introduction on RRL and shows an application to the blocks world problem.

The main strength of relational learning is that it can plan ahead for several actions. The main problem is that usually a model of the environment is required. In Q-learning, such a model can be learned. RRL is thus very well applicable to situations in which planning is required, but no model of the environment is given. Though the blocks world problem falls in this category, the cart and pendulum problem does not. Therefore RRL, though being a promising technique, does not seem very applicable to the current project.

Part IV

An initial problem analysis

Chapter 9

An initial CAP controller

This chapter describes the process of setting up a continuous RL controller for the CAP system. It consists of four sections. First, some basics of setting up an RL controller are described, like which architecture is used and which reward function is applied. After that, the set-ups of the identifier, the value function and the policy are examined. Each of these function approximators has its own section, discussing issues that arose during its development, as well as a performance evaluation and potential improvements. At the end, some ideas about proving convergence are discussed.

9-1 General CAP controller design

This section discusses some general data on the CAP controller design. First the RL controller architecture and the function approximators used are discussed. This is followed by a discussion on the reward function.

9-1-1 The controller set-up

During the development of a reinforcement learning controller for a CAP system with continuous states, the theory in chapter 5 has been used. Though a fixed time step has been used, still the theory that allows for varying time steps has been implemented. This makes it possible to adjust the simulation time step of trained controllers even after training.

As function approximators, the neural networks of chapter 4 have been used. The normalizing input and output has been applied. This improvement proved to be highly necessary. If the input and output were not normalized, then the learning performance was strongly reduced. The automatic variation of the learning rate has not been applied due to time constraints. Instead, a manually tuned fixed learning rate has been used.

9-1-2 The reward function

One of the most important things to do when designing an RL controller is to define a proper reward function. It will tell the controller what kind of results it should aspire to. The reward function should thus in some way reflect the goals of the controller. The controller goals have been defined in subsection 2-2-2. Most importantly, the controller needed to keep α and x_c near zero, with a relatively low control input.

The data that is inserted into the reward function is s_{k+1} (the new state reached) and a_k (the action used to get there). The state s_{k+1} consists of the parameters α, $\dot{\alpha}$, x_c and $\dot{x}_c$ while the action a_k equals the input force F that has been applied. The reward function that has been used, which clearly reflects the goals of the controller, is

$$r(s_{k+1}, a_k) = -r_\alpha \frac{|\alpha|}{\alpha_{max}} - r_{x_c} \frac{|x_c|}{x_{c_{max}}} - r_F \left(\frac{F}{F_{max}} \right)^2, \tag{9-1-1}$$

where r_α, r_{x_c} and r_F are coefficients. They can be used to tune the controller. If position control is very important, r_{x_c} will be high, while if it is necessary to use small input forces, r_F will be high. For the tests done in this report, only values of $r_\alpha = r_{x_c} = 1$ and $r_F = 2$ have been considered.

It must be noted that earlier versions of the RL controller did not have the last term of the reward function. (That is, $r_F = 0$.) This term was added later, for reasons that will be further discussed in subsection 9-4.

One may wonder why the F term in the above reward function is squared, while the other terms are not. There of course is a reason for that. Squaring a term will reduce its influence for small values, while increasing its influence for big values. It is important to still control α and x_c, even if they have small values. For the input force F, small values don't really matter though. The controller only needs to be discouraged to use significantly big input values F. Thus, more influence for big values is required.

A final thing to discuss is: what is the value of a crashed (terminal) state? A crash should be discouraged, so the value should be lower than the value that could be obtained by ultimately preventing the crash by applying maximum input, though it shouldn't be so low as to create significant discontinuities in the value function. A good value of a terminal state (with $|\alpha| > \alpha_{max}$ or $|x_c| > x_{c_{max}}$) would thus be

$$-\frac{1}{\ln(\gamma)} \left(-r_\alpha \frac{|\alpha|}{\alpha_{max}} - r_{x_c} \frac{|x_c|}{x_{c_{max}}} - r_F \right). \tag{9-1-2}$$

In other words, when the CAP system has crashed, it is just like the system stays forever in that situation, while applying a maximum input, and forever gets rewards based on this situation.

9-2 Developing the identifier

The next step, after choosing the reward function, was to develop the identifier. This wasn't a difficult task. A neural network with one hidden layer and 16 hidden nodes proved easily

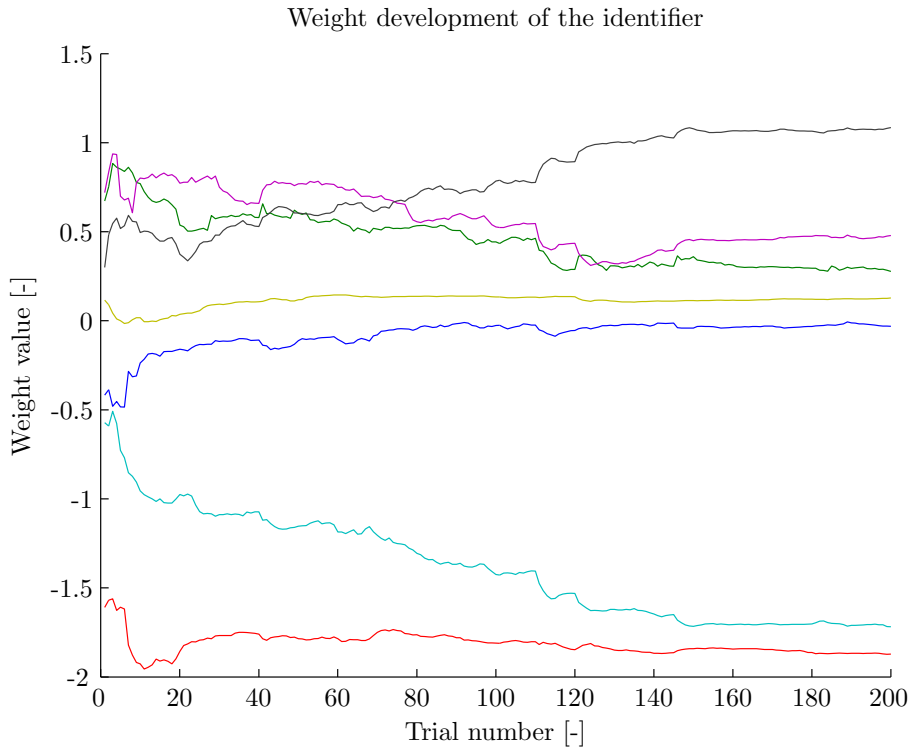

Figure 9-1: Plots of the development of seven randomly selected identifier NN weights during training. Training was done for 200 CAP simulations.

capable of the task. After 50 simulations, the identifier was adequately capable of predicting the changes in state Δs based on the current state s. Though there were some minor deviations from the exact system dynamics, these deviations were very limited – usually less than 10% of the actual change in state.

One of the reasons why the identifier was so easily capable of learning the system dynamics, is the relatively simple and near-linear behavior of the CAP system. The identifier only needs to learn the system dynamics for small values of α, and for these values the system behavior is still roughly linear. (Note that for $|\alpha| \geq 10^\circ$ the simulation is aborted.) Furthermore, the dynamics of α and $\dot{\alpha}$ do not depend on x_c or $\dot{x}_c$. This makes the relation to be learned relatively easy. Do note that this simplicity is not necessarily a downside. In an actual aircraft the dynamic behavior is also mostly linear and many parameters can also be ignored.

Besides checking the value output of the identifier, it is also important to check whether it converges to a stationary function. To check this, one can make a plot of the development of the NN weights during training. The development of seven randomly selected NN weights (out of a total of 164 weights) is shown in Figure 9-1.

It can be seen that the weights almost converge. They do have the tendency to go to a certain value, but they continue to have minor oscillations near that value. The reason for this is that the identifier cannot fully (with 100% accuracy) approximate the system dynamics. Also, different input-output combinations are continuously given to the NN for training. If these input-output combinations are a bit more concentrated in one region at a certain time, the NN adjusts to become more accurate in that region.

The small oscillations of the NN weights aren't necessarily bad. The identifier still has a sufficient performance. However, it is also easy to prevent them. Currently, the NN identifier has been trained with a constant learning rate of $\alpha = 0.1$. If this learning rate would decrease over time, according to the well-known conditions

$$\sum_{k=0}^{\infty} \alpha = \infty \qquad \text{and} \qquad \sum_{k=0}^{\infty} \alpha^2 < \infty, \tag{9-2-1}$$

then the identifier weights will also converge. However, due to time constraints, this improvement has not been implemented.

9-3 Developing the value function

Developing the value function was much more challenging. There were several issues to overcome.

9-3-1 Choosing a value function type

The first choice that had to be made was what kind of value function to use. Use a value function V (that only depends on the state s), or use an action-value function Q (that depends on both the state s and the action a)? In most RL applications using the Q-function is preferred, as it is very easy to derive the policy from it. One only needs to find the action a maximizing $Q(s, a)$ for the current state s. For discrete systems that is easily possible.

However, now the value function will be approximated using a neural network. Finding the input that maximizes the NN output is a nonlinear optimization problem and a difficult one at that. It can be solved, for example using the theory described in (de Weerdt et al., 2009), but this is computationally very intensive. It would make the algorithm too slow and is thus not practically possible. Instead, the value function V is used in this project. This has as added advantage that the value function depends on less parameters. This reduces the dimensionality and thus speeds up learning.

9-3-2 Choosing an update method

The second question to ask was which kind of updating to use: Monte Carlo methods or (bootstrapping) TD methods. Initially, TD methods have been tried, as they are known to learn faster. However, this very often resulted in divergence of the value function. The reason for this was the NN initialization. Initially, the NN is randomly initialized. In other words, the approximated value $V(s_{k+1})$ of every state s_{k+1} is unlikely to be anywhere near its actual value. When applying TD updating according to relations (5-1-5) and (5-1-6), this wrong value will cause an incorrect update. If done repeatedly, the value function can diverge. This is a problem that was also described in (Boyan & Moore, 1995).

To solve it, the Monte Carlo state-value prediction has been used. At the end of every trial, one calculates the value of every state that has been visited. (Remember that the value equals the sum of all future rewards.) These state-value combinations are then remembered,

and subsequently used to train the value function with. Since these values are all bounded, no value function divergence can occur. Instead, the value function converges to the correct values.

Though slower than TD methods, the Monte Carlo state-value prediction proved to be much more reliable. This does not mean that TD methods are fully discarded. Later on in the algorithm, when the value function has become reasonably accurate, TD methods might still be introduced. At the moment this has not been implemented in the current algorithm, but it is suggested as a possible further improvement to the learning algorithm.

9-3-3 Caching of state-value pairs

With the above improvements, the value function still didn't show convergence to the actual values. Research showed that this was because of the lack of spread of data points. When a simulation run is made, the state travels on a 'path' through the state-space. It by no means visits a representative set of all possible states. When the value function NN is trained on the corresponding state-value pairs, the whole value function is 'pulled' in the direction of these data points. This is at the cost of the value approximation of all other states.

An initial solution to solve this problem was to write a **state-value cacher**. What this does is it divides the state-space up into blocks of set width. For every block, it keeps track of which state-value pair has been encountered last. It subsequently trains the value function NN on only these state-value pairs. (This idea is similar to using a proposal distribution, as is done in (Frank et al., 2008).) This means (A) that state-value pairs in rarely frequented regions of the state-space are used for training more often and (B) that when multiple state-value pairs occur in the same block during the same simulation, some of these state-value pairs are discarded without being used for training even once.

The initial results of the state-value cacher were promising. The shape of the value function NN started to resemble the actual value function for the given policy. However, later on in the project (as will be described later), the policy changed. A changing policy also resulted in a changing value function to be approximated, as well as different regions of the state-space to be visited. The problem here was that the state-value cacher still remembered state-value pairs in old regions of the state-space that were not visited anymore. These state-value pairs were not refreshed, and so the state-value cacher still continued to use these pairs to train the value function NN. This resulted in an inability of the value function NN to converge to the value function corresponding to the new policy.

This problem can be solved in various ways, some more difficult than others. The approach chosen in this project is to keep the solution as simple as possible. (Otherwise it will be impossible to sufficiently analyze the system later on.) Hence the state-value cacher was completely discarded. Instead, the value function was trained on all state-value data points of the last n simulations (where n was initially set to 15 but later reduced to 6 for all function approximators). Furthermore, all state-value data points of these simulations were fed to the NN in a random order. By doing this, the 'pulling' of the NN towards certain data points was prevented and some additional randomness was brought into the learning algorithm. This solved the problems described earlier.

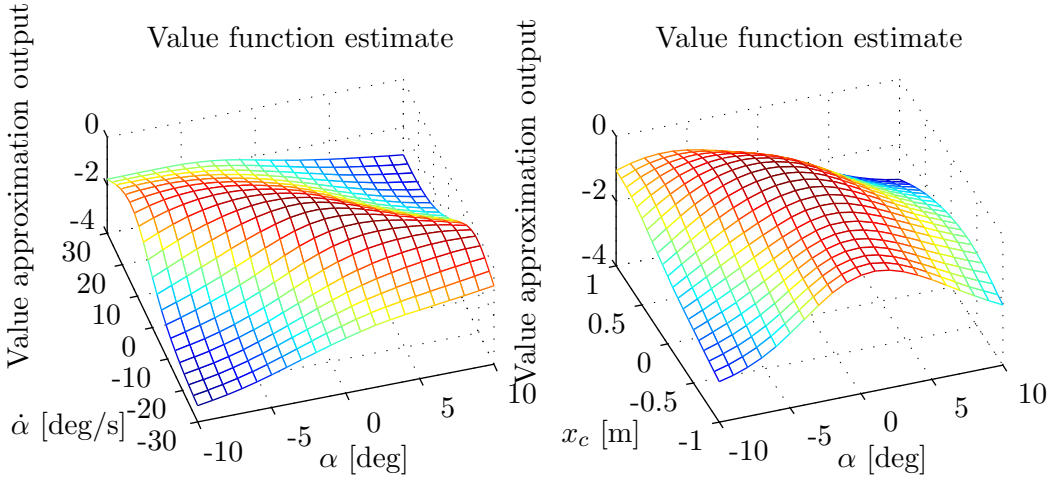

Figure 9-2: Plots of the value function after 200 CAP simulations. The left plot shows the value approximation V versus α and $\dot{\alpha}$, with $x_c = \dot{x}_c = 0$. The right plot shows it versus α and x_c, with $\dot{\alpha} = \dot{x}_c = 0$.

9-3-4 Value function evaluation

An RL controller has been trained for 200 CAP simulations, using the training method described above. This RL controller contained an identifier, a value function and a policy. (The way in which the policy was set up will be described in the next section.) It has been trained using ϵ exploration with a decreasing value of ϵ, which eventually reached zero in the last couple of simulation. This resulted in the value function shown in Figure 9-2.

First examine the left graph. Here the value function estimate is shown with respect to α and $\dot{\alpha}$. It can be seen that the highest value occurs at $\alpha \approx \dot{\alpha} \approx 0$. This value equals -0.12. Ideally (with the perfect controller and a converged value function) it would be zero, but these conditions are evidently not present yet. It can also be noted that a positive value of α combined with a negative value of $\dot{\alpha}$ (or vice versa), in the right proportions, also gives a reasonably high value. This makes sense. When the CAP system is in such a state, it is moving towards the $\alpha \approx \dot{\alpha} \approx 0$ state and will soon reach it without any required intervention.

Second, examine the right graph. It shows the value function approximation with respect to α and x_c. Again, the value function is at a maximum when $\alpha \approx x_c \approx 0$. Also a trend of high value estimates can be seen when α and x_c have opposite signs. This again makes sense. To see why, suppose that $x_c < 0$. When $\alpha > 0$, the controller (through the policy) will be giving a positive input force $F > 0$ to keep the pendulum upright. Because of this, the cart will also move to the right, thus bringing the pendulum closer to the $x_c = 0$ position, increasing the rewards given.

It can be concluded that, though the value function approximation clearly hasn't converged yet, as can also be seen from its slightly asymmetrical shape, it does give useful information with which a policy can be trained.

9-4 Developing the policy

The hardest part in the design of the RL controller was the design of the policy. The issues faced during its development will now be discussed.

9-4-1 Adding an input term to the reward

The policy has been set up using the methods described in subsection 5-2-2, together with the continuous time extension of subsection 5-3-3. The weight update law is thus given by equation (5-3-11).

It must be noted that initially the reward function (9-1-1) did not have a term with the input force (action) F. However, this resulted in the policy giving very high input values F. Divergence did not necessarily take place, but the input force F given by the policy NN far surpassed the maximum input force F_{max}. Research showed the reason for this.

Imagine the CAP system is in a state with a relatively big positive α and a big positive $\dot{\alpha}$ as well. This is a bad state with a low reward. Having a big positive α with a big negative $\dot{\alpha}$ would be a much better state with a much higher value, as this would soon bring the CAP system to $\alpha \approx \dot{\alpha} \approx 0$. If the system can, it would want to change the value of $\dot{\alpha}$ from very positive to very negative within a single very short timestep. And, when no bounds or constraints are imposed on the input force F, it will be able to. The policy will thus select a huge force F.

Imposing a hard bound on the policy is nearly impossible. The gradient descent algorithm that is used to adjust the policy π does not take into account a maximum. Instead, high values of F should simply be discouraged. A very potent way of doing this, is giving negative rewards for high input forces F. This resulted in the addition of the third term in the reward function (9-1-1).

So what is the result of this term? To find that out, examine the update law (5-3-11) again. Since the reward r now depends on the action a, the derivative $\partial r/\partial a$ is nonzero. Instead, (with $a = F$) it equals

$$\frac{\partial r}{\partial F} = -2r_F \frac{F}{F_{max}^2}. \tag{9-4-1}$$

This term acts as a 'stabilizing' term in the updating of the policy weights. When the policy is updated at a certain state s with a big value of F (magnitude-wise), the weights are adjusted such as to reduce the magnitude of F. In fact, the whole update law has become a 'tug of war' between terms, where most terms try to increase the magnitude of F to obtain desired behavior quickly, yet the term with $\partial r/\partial a$ tries to keep F contained.

Indeed, the effect of introducing the extra term into the reward function was evident. The input forces given by the policy π were not anymore in the order of several hundreds of Newtons. Instead, for all states s that occur during normal operation, they were well within the interval $[-F_{max}, F_{max}]$.

9-4-2 The effects of muscle learning

A certain degree of exploration is necessary for the policy to learn the right behavior. Part of the exploration is already provided by the fact that the CAP system is randomly initialized at every simulation run. However, also exploration for actions is necessary.

It is possible to use a fully random action from the interval $[-F_{max}, F_{max}]$ in a part ϵ of the times. Alternatively, it is possible to implement the idea of muscle learning, as described in subsection 8-2-2. Both ideas have been implemented.

During the initial stages of training, both methods had a lot of randomness. That is, the ϵ exploration method had a big value of ϵ, while muscle learning had a big variance. In these stages of training both methods tried almost every possible action, and the methods thus didn't differ much. No noticable difference between the two methods was detected.

However, as training progressed, ϵ converged to zero and the muscle learning variance decreased to almost zero. It was here that differences became noticeable. However, it was the ϵ exploration method which turned out to work better. This method was simply able to exploit the best actions and learn from it. The muscle learning action still had some noise on the input. The idea behind muscle learning was that the controller could learn the effect of these partly randomized actions. However, since the noise had a high frequency, and a slow Monte Carlo state-value prediction method was used to update the states, the controller was not able to discern any effect from all the different actions taken.

This does not imply right away that the idea of using muscle learning in RL is useless. If a faster update method (like the TD method) is used, or if the noise has a much lower frequency, then muscle learning may still be advantageous. However, due to time constraints these ideas have not been researched and will thus remain suggestions for further research.

9-4-3 The effects of the discount factor γ

During learning, several values of the decay γ have been tried. It was found that controllers with a relatively strong decay (say, a discount factor of $\gamma = 0.2$ every second) learned to keep the pendulum upright pretty quickly. Controllers with a weaker decay (say, a discount factor of $\gamma = 0.6$ every second) learned slower, yet in the end had higher performance. Controllers with very weak decay (say, a discount factor of $\gamma = 0.9$ every second) failed to learn to keep the pendulum upright well enough within the learning time.

To see why this is the case, a policy should be examined more closely. Figure 9-3 displays the policy after training on 200 simulation runs with $\gamma = 0.6$. On the left, the policy is plotted versus α and $\dot{\alpha}$. Here it can be seen that the policy is 0 if $\alpha \approx \dot{\alpha} \approx 0$, but also in the region described previously in which α and $\dot{\alpha}$ have opposite signs and their magnitudes are in just the right proportion. If α and $\dot{\alpha}$ are both positive, then a crash is imminent, and a big corrective force is required to save the system.

On the right of Figure 9-3, the policy has been plotted with respect to α and x_c. Something very important can be noticed if one looks closely at the line corresponding to $\alpha = 0$. What does the controller do when the pendulum is upright, yet the cart is in the wrong position? It turns out that F hardly depends on x_c, yet the tiny bit of dependency that actually is present says that F and x_c have the same sign. So if x_c is negative (i.e., the cart is positioned too far

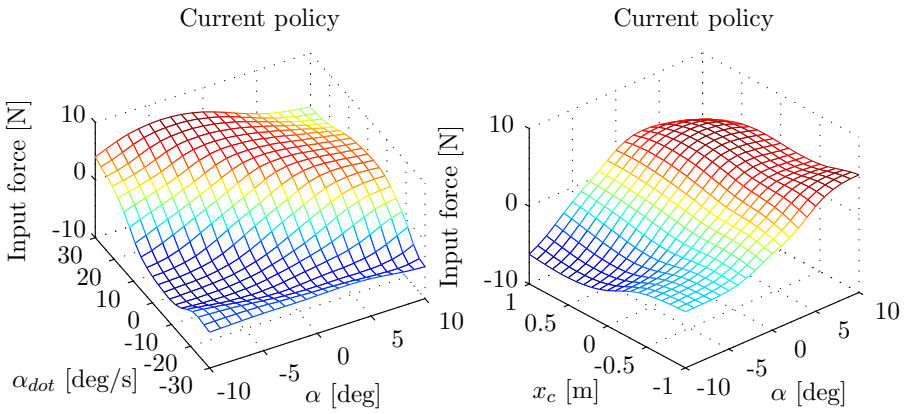

Figure 9-3: Plots of the policy after 200 CAP simulations with $\gamma = 0.6$. The left plot shows the policy π versus α and $\dot{\alpha}$, with $x_c = \dot{x}_c = 0$. The right plot shows it versus α and x_c, with $\dot{\alpha} = \dot{x}_c = 0$.

to the left), then a negative force F will be applied to the cart. This will cause the pendulum to fall to the right, which results in a positive force F, ultimately moving the pendulum to the right, increasing x_c. This is typical long-term thinking. However, as was noted previously, x_c hardly varies with F, so this tendency is present only very little.

When the controller is trained for small γ (near 0.2), things are completely different. In that case x_c and F have the opposite sign (when all other state parameters are zero). When now x_c is negative, the controller applies a positive force. This initially increases x_c, giving an improved short-term reward. However, the pendulum starts to fall to the left and a big negative input force F is required to keep the pendulum upright, thus ultimately moving the cart further to the left.

This all corresponds to the theory described on short-term versus long-term thinking in subsection 8-2-2. The key here is to start learning while only thinking short-term. Once that works, long-term thinking should be added. Thus, γ should be increased as learning progresses. This prevents the slow learning at high values of γ, as well as the low performance resulting from low values of γ. Sadly, due to time constraints, this idea has not yet been implemented and tested on an actual system yet, so no results on its actual performance can be given so far.

9-4-4 Convergence issues

The RL controller algorithm, as it has been implemented currently, still does not converge. The reason for this lies in the data points that are fed to the NNs. These are the data points of the last 6 simulation runs (in random order).

It is important to keep in mind that simulation runs are randomly initialized. It may just happen that most of these 6 simulation runs start (and stay) in one region of the state-space. The neural networks then are trained extra for this region of the state-space, at the cost of other regions. This causes the policy to vary over time.

There are several ways to solve this problem. One solution would be to change the data points that are fed to the NNs. However, which data points should be fed to the NNs instead would

then still be a topic of discussion. A better solution would instead be to vary the learning rate α. It should be decreased over time, preferably in a way satisfying relation (9-2-1). Alternatively, it can be varied in a more sophisticated way, as suggested in subsection 4-3-2. Either way will reduce the amount of variations in the policy during subsequent training, and possibly also ensure convergence, maybe even to a local optimum, though no certainty on this can be given yet. Also, due to time constraints, a varying learning rate has not yet been implemented, so no definite claims can be made as to the effects it would have.

9-4-5 Policy evaluation

A plot of the policy has already been shown. But does the policy also work in controlling the CAP system? To do that, a basic simulation using the resulting policy has been run. The result is shown in Figure 9-4.

Figure 9-4 shows that the controller manages to keep the pendulum upright at $\alpha = 0$. This is done without an excessive input force F. Only during the first few timesteps of the simulation run are significant input forces required. Also, the controller is capable of keeping x_c within the specified bounds. However, x_c is not being kept at 0. The reason for this is, as was discussed previously, that the controller doesn't think that long-term. If x_c needs to be controlled adequately, then γ needs to be increased.

Concluding, a basic RL controller has already been written that can reasonably control the CAP system. However, several improvements still need to be made before all requirements are met. And when that has been done, the most difficult step still awaits: finding out whether the controller always converges to a solution, and whether this solution is a local or a global optimum.

9-5 Ideas on proving convergence

Consider the CAP controller as has been designed in this chapter. This controller has plenty of issues. What should the value of γ be? What should be done with the learning rate α? In what way should the state-space be explored? These are all complicated issues.

Ideas to solve all these issues have been presented. However, there are still several downsides.

- Solving all these issues will cost a lot of time and make the algorithm much more complex.
- Once the improvements have been made, the algorithm may (or may not) always converge, but the algorithm complexity will make it hard to prove convergence.
- Even if convergence has been established, it does not imply that this is convergence to the optimal (or any useful) policy. Of course improvements can be made to encourage convergence to a useful policy (like more exploration), but this still does not guarantee global optimality.
- Ensuring global optimality will be hard, since it requires three neural networks to be optimized together. All neural networks have huge numbers of weights, and finding the global optimum in any practically viable way seems like an impossible task.

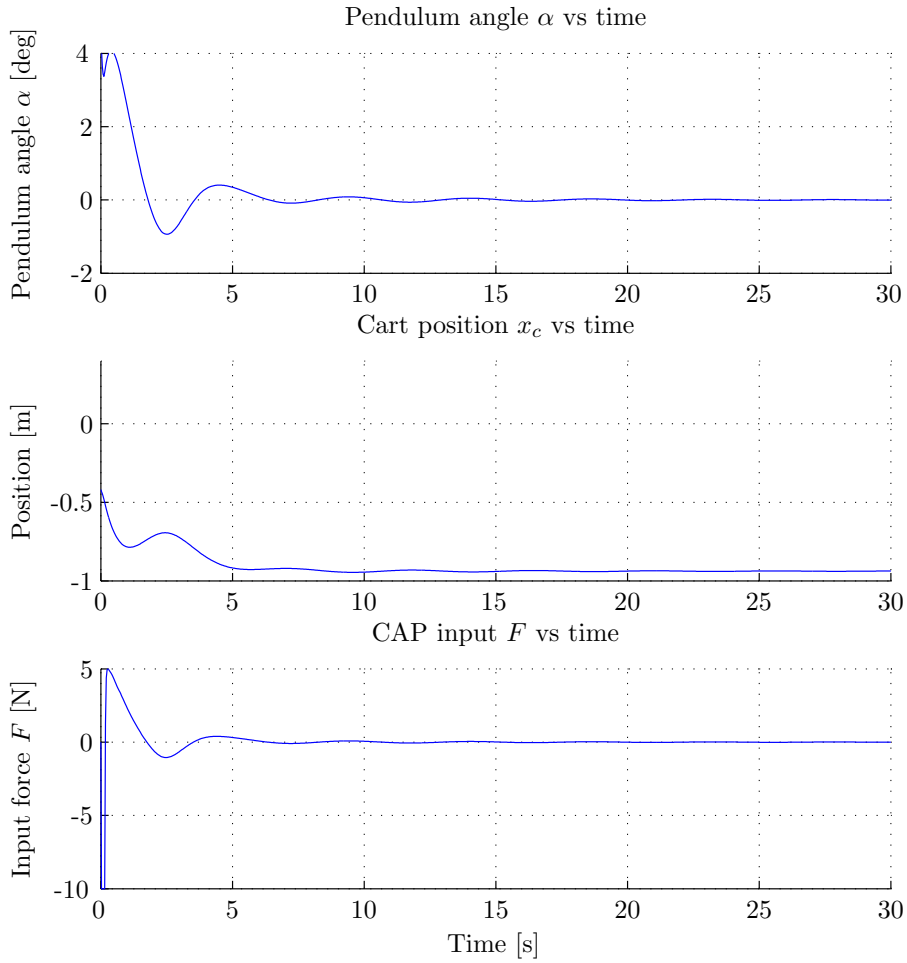

Figure 9-4: Plots of a simulation run performed from a random state by an RL controller that was trained on 200 CAP simulations with $\gamma = 0.6$.

The above points don't imply that it's impossible to use neural networks to design an RL algorithm that always converges to the global optimum. They do imply that expanding the current RL algorithm to an algorithm with globally optimal convergence is very complicated, and is unlikely to be obtained within the scope of this project. Therefore, alternative approaches need to be considered on obtaining proven convergence to the global optimum.

Chapter 10

Combining Q-learning with interval analysis

One of the goals of this report is to check whether interval analysis can be successfully combined with reinforcement learning, and subsequentially whether the resulting algorithm can be proven to converge. Combining RL with IA has never been done before in literature. So a preliminary indication on how this can be done is required.

This chapter considers a combination of Q-learning and IA, called the **interval Q-learning** algorithm. Though only an extension of discrete Q-learning (without function approximation) as described in subsection 3-3-3 is considered here, keep in mind that IA can be applied in a similar way to other RL algorithms. After this extension has been described, it will be analyzed and its advantages and disadvantages will be examined. Later on, in chapter 11, the discrete interval Q-learning algorithm will be expanded to a continuous domain.

10-1 The interval Q-learning algorithm

In interval analysis, numbers are replaced by intervals that give bounds on these numbers. In Q-learning, the numbers to be considered are the values $Q(s,a)$. This gives rise to the idea to replace the Q-values by intervals. That is in fact the basis of the interval Q-learning algorithm.

To start off, the Q-function is initialized for all states s and actions a as an interval from the minimum value to the maximum value. So,

$$Q(s,a) = [Q_{min}, Q_{max}] \tag{10-1-1}$$

for every s, a. This step must ensure that the optimal value of the Q-function, denoted by the (degenerate) function Q^*, is inside the interval function $Q(s,a)$.

Next, an update law is required. Suppose that the agent is in state s_k, chooses an action a_k and winds up in state s_{k+1}. In normal Q-learning, update law 3-3-5 is used. This law should

now be adjusted to update both the minimum $\underline{Q}(s_k, a_k)$ and the maximum $\overline{Q}(s_k, a_k)$ possible value of $Q^*(s_k, a_k)$.

To find the minimum $\underline{Q}(s_k, a_k)$, one should adopt a conservative approach. One ignores the possible maxima of $Q(s_{k+1}, a)$ and only looks at all possible minima. One then takes the highest of these minima to update $\underline{Q}(s_k, a_k)$. In an equation, this becomes

$$\underline{Q}_{new}(s_k, a_k) \leftarrow \underline{Q}(s_k, a_k) + \alpha \left(r_{k+1} + \gamma \max_a \underline{Q}(s_{k+1}, a) - \underline{Q}(s_k, a_k) \right). \tag{10-1-2}$$

One can see this update law as 'Searching for the minimum value which, in the worst case, can always be obtained.' Finding the maximum $\underline{Q}(s_k, a_k)$ goes similar, but now one adopts an optimistic approach. One only considers the possible maxima of $Q(s_{k+1}, a)$. This gives

$$\overline{Q}_{new}(s_k, a_k) \leftarrow \overline{Q}(s_k, a_k) + \alpha \left(r_{k+1} + \gamma \max_a \overline{Q}(s_{k+1}, a) - \overline{Q}(s_k, a_k) \right). \tag{10-1-3}$$

This update law can be seen as 'Finding the maximum value which, when everything goes perfectly, might be obtained.' Note that the interval with the maximum lower limit $\underline{Q}(s_{k+1}, a)$ does not necessarily correspond to the interval with the maximum upper limit $\overline{Q}(s_{k+1}, a)$, and it is thus not possible to combine these two update laws in one relation.

It can also be noted (through a simple numerical example) that the new interval $Q_{new}(s_k, a_k)$ isn't always a subset of the old interval $Q_{old}(s_k, a_k)$. However, the old bounds $Q_{old}(s_k, a_k)$ of course still hold. An improvement can thus be obtained when one takes the intersection of the two intervals. Thus, after applying the above update laws, one also applies

$$Q(s_k, a_k) \leftarrow Q_{new}(s_k, a_k) \cap Q_{old}(s_k, a_k). \tag{10-1-4}$$

10-2 Analyzing the algorithm

To analyze the algorithm described above, one can perform tests, or one can perform a theoretical analysis. In this section, the latter will be done. It turns out that for deterministic environments it can be proven that the algorithm works. That will be done in this section.

10-2-1 Bounding of the optimal value function

First one may ask: will the interval $Q(s, a)$ always contain the optimal value $Q^*(s, a)$ for every s, a? This is indeed the case, as will be proven now.

Initially, $Q(s, a)$ contains $Q^*(s, a)$ for every state s, a. (This follows from the algorithm initialization.) Now suppose that there will be an update step in which $Q(s, a)$ will not contain $Q^*(s, a)$ anymore for some combination s, a. Consider the first time k that this happens. Refer to the Q-function before this update as $Q_{old}(s, a)$ and the Q-function after the update as $Q_{new}(s, a)$. This implies that $Q^*(s_k, a_k) \notin Q_{new}(s_k, a_k)$ for s_k, a_k, while $Q^*(s, a) \in Q_{old}(s, a)$ for all s, a. This directly implies that either $Q^*(s_k, a_k) < \underline{Q}_{new}(s_k, a_k)$ or $Q^*(s_k, a_k) > \overline{Q}_{new}(s_k, a_k)$.

First consider the case where $Q^*(s_k, a_k) < \underline{Q}_{new}(s_k, a_k)$. The Bellman optimality condition (for a deterministic environment) states that

$$Q^*(s_k, a_k) = r_{k+1} + \gamma \max_a Q^*(s_{k+1}, a), \tag{10-2-1}$$

where state s_{k+1} follows from taking action a_k at state s_k. Now note the three inequalities

$$\begin{aligned} \underline{Q}_{new}(s_k, a_k) &> Q^*(s_k, a_k), & (10\text{-}2\text{-}2)\\ Q^*(s_k, a_k) &\geq \underline{Q}_{old}(s_k, a_k), & (10\text{-}2\text{-}3)\\ Q^*(s_k, a_k) = r_{k+1} + \gamma \max_a Q^*(s_{k+1}, a) &\geq r_{k+1} + \gamma \max_a \underline{Q}_{old}(s_{k+1}, a). & (10\text{-}2\text{-}4) \end{aligned}$$

Thus, for every $\alpha \in [0, 1]$, it also holds that

$$\begin{aligned} \underline{Q}_{new}(s_k, a_k) &> (1-\alpha)\underline{Q}_{old}(s_k, a_k) + \alpha(r_{k+1} + \max_a \underline{Q}_{old}(s_{k+1}, a)) & (10\text{-}2\text{-}5)\\ &= \underline{Q}_{old}(s_k, a_k) + \alpha\left(r_{k+1} + \gamma \max_a \underline{Q}_{old}(s_{k+1}, a) - \underline{Q}_{old}(s_k, a_k)\right) & (10\text{-}2\text{-}6) \end{aligned}$$

However, this is in complete violation with update law (10-1-2), which claims equality. Thus there is a contradiction. It cannot occur that $Q^*(s_k, a_k) < \underline{Q}_{new}(s_k, a_k)$. With a similar proof, it can be shown that $Q^*(s_k, a_k) > \overline{Q}_{new}(s_k, a_k)$ is also impossible. Concluding, it cannot occur that at some time, for some combination s, a, the interval $Q(s, a)$ does not contain $Q^*(s, a)$. Instead, the interval $Q(s, a)$ will always contain $Q^*(s, a)$.

It is interesting to note that, in the above proof, α can take any value in the interval $[0, 1]$. This suggests setting α simply equal to 1, which would speed up convergence.

10-2-2 Converging to the optimal value function

A second question to ask is: will the interval Q-learning algorithm converge? The quick answer here is yes, if all state-action pairs are continually visited. Due to the nesting relation (10-1-4), the subsequent intervals $Q(s, a)$ are nested. As was mentioned in subsection 6-3-3, this proves that the interval sequence desribed by the continuous updating of $Q(s, a)$ converges. However, this does not imply convergence to the degenerate value $Q^*(s, a)$. Instead, it may also converge to a (possibly wide) interval. Now it will be proven that the algorithm indeed converges to degenerate values.

Consider the case where the algorithm has converged. Thus, updating does not change the value of $Q(s, a)$ for any combination s, a. Now examine the width $w(Q(s_k, a_k))$ at some time k. It equals

$$w(Q(s_k, a_k)) = \overline{Q}(s_k, a_k) - \underline{Q}(s_k, a_k). \tag{10-2-7}$$

Working this out using the update relations (10-1-2) and (10-1-3) gives

$$w(Q(s_k, a_k)) = (1-\alpha)w(Q(s_k, a_k)) + \alpha\gamma\left(\max_a \overline{Q}(s_{k+1}, a) - \max_a \underline{Q}(s_{k+1}, a)\right), \tag{10-2-8}$$

or equivalently (if $\alpha \in (0, 1]$),

$$w(Q(s_k, a_k)) = \gamma\left(\max_a \overline{Q}(s_{k+1}, a) - \max_a \underline{Q}(s_{k+1}, a)\right). \tag{10-2-9}$$

Examine the quantity on the right. It can be shown to satisfy

$$\max_a \overline{Q}(s,a) - \max_a \underline{Q}(s,a) \leq \max_a \left(\overline{Q}(s,a) - \underline{Q}(s,a)\right) = \max_a w(Q(s,a)) \tag{10-2-10}$$

for any state s. (To see why, consider the action a' that maximizes $\overline{Q}(s,a')$ and consider which potential values $\underline{Q}(s,a')$ might have.) Therefore,

$$w(Q(s_k,a_k)) \leq \gamma \max_a w(Q(s_{k+1},a)). \tag{10-2-11}$$

Now consider the combination s_k, a_k that maximizes $w(Q(s,a))$. (If there are multiple, take any combination.) If $\gamma < 1$, then the above statement says that there is some other interval $Q(s_{k+1},a)$ that is wider than $Q(s_k,a_k)$. This cannot be, as s_k, a_k was the widest interval. This implies that $w(Q(s_k,a_k)) = 0$. Thus, if $\gamma < 1$, the algorithm always converges to a degenerate solution.

Concluding, it has been proven that the intervals $Q(s,a)$ in the interval Q-learning algorithm always converge to the optimal values $Q^*(s,a)$. This is a promising start of combining RL with IA. However, a careful reader might have noticed that the algorithm basically executes two simultaneous versions of Q-learning: one with very low starting values and one with very high starting values. Since Q-learning always converges, it is evident that the algorithm described above also converges. Thus, so far nothing new has been developed. However, the question does arise whether it is possible to extend this algorithm to the world of continuous states and actions.

10-3 Issues in the extension to a continuous domain

When attempting to extend the interval Q-learning for continuous states, several problems arise. These problems will be discussed in this section.

The problem with continuous states and actions is that one needs an infinite amount of intervals to approximate Q. This is of course impossible, so instead one can use a function approximator for $\underline{Q}$ and a function approximator for $\overline{Q}$. This poses some risks though. Imagine that the approximation of $\underline{Q}(s,a)$ is adjusted. This will also adjust the approximations of $\underline{Q}(s',a')$ for s' and a' close to s and a. Often these adjustments are warranted, but it may occur that this is not the case. If that happens, then the guarantee that $Q^*(s',a') \in Q(s',a')$ is gone, and the algorithm may thus not find the optimal value function.

A second problem arises in deriving a policy. Previously in this report, it has been argued that it is computationally impractical to find the action a that maximizes $Q(s,a)$ for some state s. As a result, it was posed to use a value function V instead, and derive a policy through the derivative $\partial V/\partial s$. However, if instead an interval value function $V = [\underline{V}, \overline{V}]$ is used, such a derivative does not readily exist. Instead, one can take (for example) the derivative of the midpoint $\partial m(V)/\partial s$ or the derivative of the maximum $\partial \overline{V}/\partial s$, but neither of these choices seem to have an easy and logical justification. Alternatively, one can also set up the policy as an interval function approximation, and subsequently use $\partial \overline{V}/\partial s$ to update $\overline{\pi}$, while $\partial \underline{V}/\partial s$ is used to update $\underline{\pi}$, but this approach is also not without further issues.

Finally, the convergence proof of the previous section only holds for deterministic environments. It is not clear yet whether the interval Q-learning algorithm also works for stochastic

environments (either discrete or continuous). The stochastic environment makes the proof that always $Q^*(s,a) \in Q(s,a)$ invalid. In fact, if the environment gives a particularly high or low reward for one time, it may occur that $Q^*(s,a) \notin Q(s,a)$ for some s, a. This is a problem.

To get rid of it, one needs to delete the nesting step at the end of the interval Q-learning iteration. Though this would solve the problem, it basically turns the whole interval Q-learning algorithm into two simultaneous ordinary Q-learning algorithms, and without the guarantee that $Q^*(s,a) \in Q(s,a)$ there are no added benefits of using two Q-learning algorithms instead of just one. Thus, so far the interval Q-learning algorithm does not provide any advantages for stochastic environments.

These problems are all detrimental for the further expansion of the interval Q-learning algorithm. A lot of further research is required to solve them. Later on in this report, in chapter 11, this will in fact be done, and most of these problems will be solved. An extension of the interval Q-learning algorithm to continuous systems will be set up, which has proven convergence for continuous deterministic systems.

Part V

The continuous interval Q-learning algorithm

Chapter 11

The theoretical set-up of the continuous Q-learning algorithm

Earlier on, in chapter 10, the interval Q-learning algorithm was set up. This chapter will expand this algorithm to the continuous domain. First, section 11-1 will set up the basic theory of the algorithm, followed by the algorithm itself. Then section 11-2 will evaluate the algorithm.

11-1 Setting up the continuous interval Q-learning algorithm

This section will discuss the main ideas behind the continuous Q-learning algorithm. Once the main ideas have been mentioned, they are combined into an algorithm: the continuous interval Q-learning algorithm.

11-1-1 The fundamental assumption

Consider a continuous reinforcement learning problem. In literature, there are many controllers that solve such problems. They all use function approximators. These function approximators all use data on a state s to update the values of nearby states s'. The algorithms thus assume that nearby states have similar values. However, almost all articles fail to mention this assumption. It is one that is silently assumed to be true. And that is strange, since the assumption is one on which the entire algorithm is based.

In this report, things are done differently. The assumption that nearby states have nearby values is not only mentioned, it's also quantified. This quantification will turn out to be the key to setting up a reinforcement learning algorithm with proven global convergence. Without it, it will be very hard, if not impossible, to prove globally optimal convergence for continuous reinforcement learning controllers.

The main RL value assumption

The slope of the value function Q is assumed to have a certain known maximum value or bound dQ/ds_i or dQ/da_j for every state parameter s_i and action parameter a_j.

How these maximum slopes are found is something that will be briefly discussed later in this report, in subsection 12-6-3. For now it is simply assumed that decent upper bounds are known for the derivatives with respect to each parameter. Do note that the quantities dQ/ds_i and dQ/da_j do not indicate actual derivatives but only bounds on the absolute value of these derivatives.

11-1-2 The application to interval Q-learning

The above assumption can be used to extend the interval Q-learning algorithm to the continuous domain. Imagine that at some time k the RL agent is in the state vector $\mathbf{s_k}$ and chooses the action vector $\mathbf{a_k}$. This brings him to a state $\mathbf{s_{k+1}}$ with reward r_{k+1}. Also assume that bounds $Q_{min}(\mathbf{s},\mathbf{a})$ and $Q_{max}(\mathbf{s},\mathbf{a})$ are known for all states $\mathbf{s}$ and actions $\mathbf{a}$. Now new bounds $Q_{min}(\mathbf{s_k},\mathbf{a_k})$ and $Q_{max}(\mathbf{s_k},\mathbf{a_k})$ can be obtained for the state-action combination $\mathbf{s_k},\mathbf{a_k}$, according to relations (10-1-2) and (10-1-3), just like in the normal interval Q-learning algorithm. For ease of reading, these relations are repeated here.

$$\underline{Q}_{new}(\mathbf{s_k},\mathbf{a_k}) \leftarrow \underline{Q}(\mathbf{s_k},\mathbf{a_k}) + \alpha\left(r_{k+1} + \gamma \max_{\mathbf{a}} \underline{Q}(\mathbf{s_{k+1}},\mathbf{a}) - \underline{Q}(\mathbf{s_k},\mathbf{a_k})\right), \quad (11\text{-}1\text{-}1)$$

$$\overline{Q}_{new}(\mathbf{s_k},\mathbf{a_k}) \leftarrow \overline{Q}(\mathbf{s_k},\mathbf{a_k}) + \alpha\left(r_{k+1} + \gamma \max_{\mathbf{a}} \overline{Q}(\mathbf{s_{k+1}},\mathbf{a}) - \overline{Q}(\mathbf{s_k},\mathbf{a_k})\right). \quad (11\text{-}1\text{-}2)$$

Note that nesting is also again applied. That is, if $\underline{Q}_{new}(\mathbf{s_k},\mathbf{a_k}) \leq \underline{Q}_{old}(\mathbf{s_k},\mathbf{a_k})$, it is not updated, and a similar thing holds for the upper bound.

However, more can be done than just updating $Q(\mathbf{s_k},\mathbf{a_k})$. The interval $Q(\mathbf{s_k},\mathbf{a_k})$ has been updated. This also means that the interval $Q(\mathbf{s'},\mathbf{a'})$ can be updated for states $\mathbf{s'}$ and actions $\mathbf{a'}$ near $\mathbf{s_k}$ and $\mathbf{a_k}$. For this, the slopes dQ/ds_i and dQ/da_j are used. Given the new value of $Q_{min}(\mathbf{s_k},\mathbf{a_k})$, the bounds for any other state $\mathbf{s}$ and action $\mathbf{a}$ can be updated as

$$Q_{min}(\mathbf{s},\mathbf{a}) = Q_{min}(\mathbf{s_k},\mathbf{a_k}) - \sum_{i=1}^{n_s} \frac{dQ}{ds_i}\left|s_i - (s_k)_i\right| - \sum_{i=1}^{n_a} \frac{dQ}{da_i}\left|a_i - (a_k)_i\right|, \quad (11\text{-}1\text{-}3)$$

with n_s the number of states and n_a the number of actions. Also, $(s_k)_i$ is the i^{th} parameter of state $\mathbf{s_k}$. It directly follows from the mean value theorem that these bounds are certain. As long as the main RL value assumption holds, they cannot be violated.

To simplify notation, define $D = n_s + n_a$, where D will be known as the **dimension** of the problem. Also merge the state $\mathbf{s}$ and the action $\mathbf{a}$ into one vector $\mathbf{x}$. (So $\mathbf{x}^T = [\mathbf{s}^T, \mathbf{a}^T]$. This means that $x_i = s_i$ when $i \leq n_s$ and $x_i = a_{i-n_s}$ when $i > n_s$.) This simplifies the above relation to

$$Q_{min}(\mathbf{x}) = Q_{min}(\mathbf{x_k}) - \sum_{i=1}^{D} \frac{dQ}{dx_i}\left|x_i - (x_k)_i\right|. \quad (11\text{-}1\text{-}4)$$

Similarly, the maximum is given by

$$Q_{max}(\mathbf{x}) = Q_{max}(\mathbf{x_k}) + \sum_{i=1}^{D} \frac{dQ}{dx_i}\left|x_i - (x_k)_i\right|. \quad (11\text{-}1\text{-}5)$$

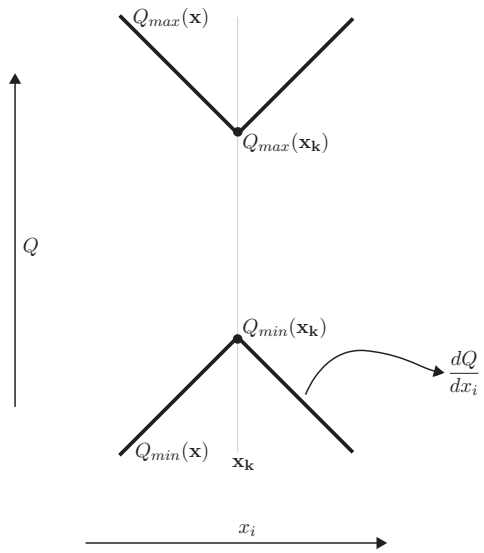

Figure 11-1: New bounds during one update step of the continuous interval Q-learning algorithm. The bounds take the shape of pyramids with as slopes the maximum possible slopes dQ/dx_i.

This idea is visualized for the one-dimensional case (with $D = 1$) in figure 11-1. If the problem is multi-dimensional, then the bounds will not take the form of triangles but of pyramids. These triangles will thus be referred to as **update pyramids**.

11-1-3 Multiple updates

The effect of one update has been shown in figure 11-1. What happens when multiple updates are done, one after the other? In that case, multiple triangles/pyramids will appear. This is visualized in figure 11-2. All these pyramids give bounds for Q. How does one know which bound to take?

That question is in fact easily answered. All the bounds that are given hold. They are all true. Hence, one can simply take the most narrow bound possible. This bound will then still hold. So, when one wants to determine $Q_{min}(\mathbf{x})$ for some state-action point $\mathbf{x}$, one looks at the value of all possible minimum-pyramids at the point $\mathbf{x}$ and selects the highest value as the lower bound $Q_{min}(\mathbf{x})$. Similarly, for the upper bound, the lowest value of all maximum-pyramids is selected for $Q_{max}(\mathbf{x})$.

11-1-4 The best exploration strategy

The goal of the interval Q-learning algorithm is to narrow the bounds on the Q-values. Preferably, the new width $w(Q_{new}(\mathbf{x_k}))$ of the state-action point $\mathbf{x_k}$ is much smaller than the old width $w(Q_{old}(\mathbf{x_k}))$. Hence, the reduction factor $w(Q_{new}(\mathbf{x_k}))/w(Q_{old}(\mathbf{x_k}))$ should be minimized. How can this best be done?

First examine the width $w(Q_{new}(\mathbf{x_k}))$ more closely. For the discrete interval Q-learning algorithm, a bound was given by relation (10-2-11). This relation still holds. Hence,

$$w(Q_{new}(\mathbf{x_k})) \leq \gamma \max_{\mathbf{a}} w(Q(\mathbf{s_{k+1}}, \mathbf{a})). \tag{11-1-6}$$

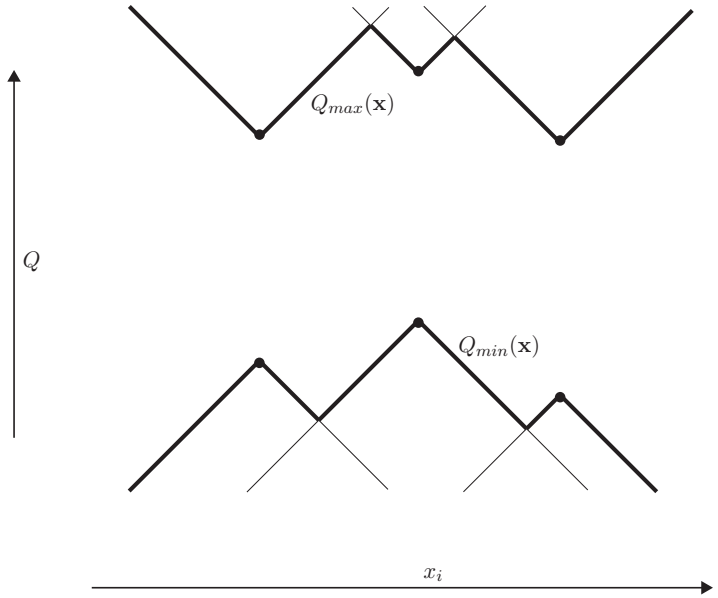

Figure 11-2: New bounds after multiple update steps of the continuous interval Q-learning algorithm. Where the bounds overlap, the most narrow bound is selected.

However, $w(Q(\mathbf{s_{k+1}}, \mathbf{a}))$ depends on $\mathbf{s_{k+1}}$, which depends on the environment. Since no knowledge is assumed on the environment, nothing can be derived on $\mathbf{s_{k+1}}$. Luckily, this problem can be worked around.

Suppose that it is possible to put the RL agent in any state $\mathbf{s_k}$. One can then set $\mathbf{x_k}$ to the state-action combination with maximum width. That is,

$$\mathbf{x_k} = \arg\max_{\mathbf{x}} w(Q(\mathbf{x})). \tag{11-1-7}$$

This ensures that $\max_{\mathbf{a}} w(Q(\mathbf{s_{k+1}}, \mathbf{a})) \leq w(Q_{old}(\mathbf{x_k}))$ for any possible state $\mathbf{s_{k+1}}$. Hence the reduction factor of $w(Q(\mathbf{x_k}))$ is always at most a factor γ.

This strategy of taking the state and the action with the maximum width is called the **best explorative strategy** for the continuous interval Q-learning algorithm. It is also a strategy that intuitively makes sense. The width of the interval $Q(\mathbf{x})$ can be seen as the 'uncertainty' in the value Q of a point $\mathbf{x}$. When one always explores the point with the highest width, and thus the highest uncertainty, the most can be learnt.

11-1-5 The solution to the exploration vs. exploitation dilemma

The best exploration strategy of the previous subsection can be expanded further. Suppose that the agent is in some state $\mathbf{s}$ and can choose between two actions $\mathbf{a_1}$ and $\mathbf{a_2}$, but he knows that $\overline{Q}(\mathbf{s}, \mathbf{a_1}) \leq \underline{Q}(\mathbf{s}, \mathbf{a_2})$. In other words, he knows that choosing action $\mathbf{a_2}$ will always result in a better value than choosing action $\mathbf{a_1}$. In this case, the agent can be fully certain that action $\mathbf{a_1}$ is not part of the optimal strategy. This action (as evaluated from the current state) can thus be fully ignored, even if $w(Q(\mathbf{s}, \mathbf{a_1})) > w(Q(\mathbf{s}, \mathbf{a_2}))$. This process is known as **rejecting certainly sub-optimal actions**.

It is interesting to note that the rejection of certainly sub-optimal actions is a solution to the exploration vs. exploitation dilemma. This dilemma is as old as reinforcement learning itself, but if the best exploration strategy is applied together with rejecting certainly sub-optimal actions, the dilemma is solved. The agent will recognize it when certain actions are not part of the optimal strategy and ignore those actions in the remainder of the algorithm run. It will thus automatically switch from exploration towards exploitation, without any risk that he might not find the optimal strategy.

Finally, it must be noted that in may practical RL applications the effectiveness of the rejection of certainly sub-optimal actions often remains limited. In a lot of continuous RL problems, there is a very small timestep, and any change in the action Δa will thus only have a very small change in the value ΔQ. As long as $w(Q)$ is significantly bigger than ΔQ, no certainly sub-optimal actions can be rejected. However, in problems where different actions a do have a significant effect on the value Q, the rejection of certainly sub-optimal actions will be very useful.

11-1-6 The continuous interval Q-learning algorithm

Now all the ideas are present to set up the continuous interval Q-learning algorithm. This algorithm is given below.

Algorithm 1 – The continuous interval Q-learning algorithm

Require: Ranges $R_i = [x_{i_{min}}, x_{i_{max}}]$ for the parameters.
Require: Upper bounds dQ/dx_i for the value function slopes.
Require: The minimum Q_{min} and maximum Q_{max} of the value function.
for $k = 1 \rightarrow n_{iterations}$ **do**
Select the point $\mathbf{x_k}$ that maximizes $w(Q(\mathbf{x}))$.
Split $\mathbf{x_k}$ up into a state $\mathbf{s_k}$ and an action $\mathbf{a_k}$. From state $\mathbf{s_k}$, execute action $\mathbf{a_k}$.
Calculate the interval Q_k according to

$$Q_k = [r_{k+1} + \gamma \max_{\mathbf{a}} \underline{Q}(\mathbf{s_{k+1}}, \mathbf{a}), r_{k+1} + \gamma \max_{\mathbf{a}} \overline{Q}(\mathbf{s_{k+1}}, \mathbf{a})] \tag{11-1-8}$$

and use it to update the Q-function.
end for

There is one part missing in the above algorithm. How does the function $Q(\mathbf{x})$ work? That part is discussed now.

Algorithm 2 – The exact interval Q-function

Require: A list of intervals Q_k, with $1 \leq k \leq n_{iterations}$.
Require: The point $\mathbf{x}$ (with $\mathbf{x}^T = [\mathbf{s}^T, \mathbf{a}^T]$) for which $Q(\mathbf{x})$ needs to be found.
Start off with $Q_{result} = [Q_{min}, Q_{max}]$.
for $k = 1 \rightarrow n_{iterations}$ **do**
Find the interval $Q_k(\mathbf{x})$ corresponding to the point $\mathbf{x}$ on the pyramid of iteration k:

$$Q_k(\mathbf{x}) = \left[\underline{Q}_k - \sum_{i=1}^{D} \frac{dQ}{dx_i} \left| x_i - (x_k)_i \right|, \overline{Q}_k + \sum_{i=1}^{D} \frac{dQ}{dx_i} \left| x_i - (x_k)_i \right| \right]. \tag{11-1-9}$$

Set $Q_{result} \leftarrow Q_{result} \cap Q_k(\mathbf{x})$.

end for
return Q_{result}.

Algorithm 1, together with algorithm 2 for finding Q-value intervals, makes up the continuous interval Q-learning algorithm. It can be proven to converge, as will be done in the upcoming section.

11-2 Analysing the continuous interval Q-learning algorithm

This section analyses the continuous interval Q-learning algorithm. It will prove its convergence and provide estimates on its performance.

11-2-1 Defining parameters

It is quite easy to see why the continuous interval Q-learning function converges. Consider the area (for a one-dimensional problem) or the volume (for a multi-dimensional problem) between $\underline{Q}$ and $\overline{Q}$. This volume decreases at every iteration. Since it cannot be negative, it therefore must converge. And, in a way very similar to the way shown in subsection 10-2-2, the volume can be shown to converge to zero. Furthermore, since the volume will always contain the actual Q-function Q^*, it must converge to a volume containing Q^*.

What would be more interesting though, is how fast the algorithm converges. That will be the subject of this section. Before that is done, some definitions need to be made. Most of these parameter definitions have been visualized in figure 11-3.

First of all, the **range** R_i is the interval containing all possible values of the parameter x_i. Thus, $R_i = [x_{i_{min}}, x_{i_{max}}]$. Secondly, the **parameter area** is defined as the area spanned by all parameters x_i. So,

$$A = w(R_1)w(R_2)\dots w(R_D) = \prod_{i=1}^{D} w(R_i). \tag{11-2-1}$$

Note that this is a D-dimensional area. Thirdly, the **remaining volume** V is defined as

$$V = \int_{R_1}\int_{R_2}\dots\int_{R_D} w(Q(\mathbf{x}))\, dx_D \dots dx_2\, dx_1, \tag{11-2-2}$$

where the integration is over all parameters x_i. Fourthly, there is the **average width** w_{av},

$$w_{av} = \frac{V}{A}. \tag{11-2-3}$$

There are also the average minimum value $\underline{Q}_{av}$ and average maximum value $\overline{Q}_{av}$, respectively defined as

$$\underline{Q}_{av} = \frac{1}{A}\int_{R_1}\dots\int_{R_D} \underline{Q}(\mathbf{x})\, dx_D \dots dx_1, \tag{11-2-4}$$

$$\overline{Q}_{av} = \frac{1}{A}\int_{R_1}\dots\int_{R_D} \overline{Q}(\mathbf{x})\, dx_D \dots dx_1. \tag{11-2-5}$$

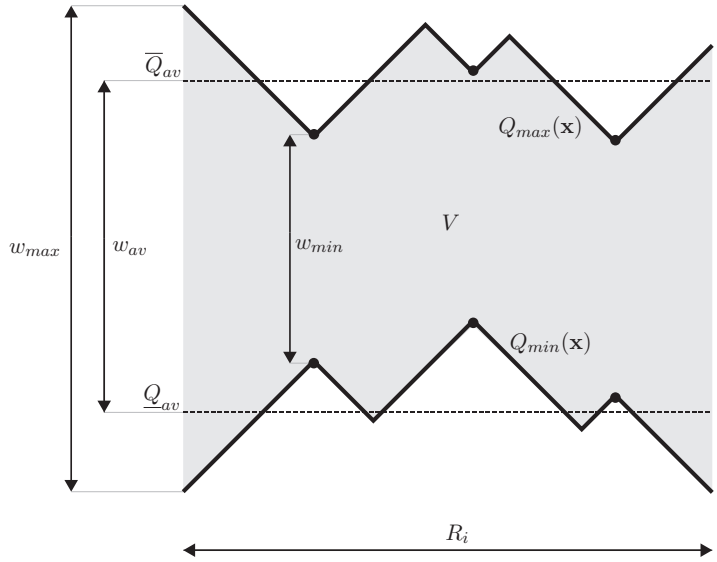

Figure 11-3: Parameters for the continuous interval Q-learning algorithm. (Note that, for a one-dimensional problem as shown here, $A = w(R_1)$.)

Note that $w_{av} = \overline{Q}_{av} - \underline{Q}_{av}$. Finally, the minimum and maximum widths are respectively defined as

$$w_{min} = \min_{\mathbf{x}} w(Q(\mathbf{x})), \tag{11-2-6}$$

$$w_{max} = \max_{\mathbf{x}} w(Q(\mathbf{x})). \tag{11-2-7}$$

11-2-2 Bounding the change in volume

Consider one iteration of the continuous interval Q-learning algorithm. How much will this iteration cause the volume to decrease? That is in fact something that cannot be exactly determined. However, it is possible to give a minimum bound on the reduction in volume.

Consider an update at some point $\mathbf{x_k}$. Such an update is visualized in figure 11-4. During this update, a new interval $Q(\mathbf{x_k})$ has been found with width $w_{update} = w(Q_{new}(\mathbf{x_k}))$. According to the continuous interval Q-learning algorithm, the point $\mathbf{x_k}$ was the point with the largest width everywhere. Hence, $w_{max} = w(Q_{old}(\mathbf{x_k}))$. It now follows that

$$w_{update} = w(Q_{new}(\mathbf{x_k})) \leq \gamma w(Q_{old}(\mathbf{s_{k+1}})) \leq \gamma w_{max}. \tag{11-2-8}$$

This thus bounds the width of the maximum update. However, to find the reduction in volume, it is more important to find the widths of the pyramids $w_{pyramid}$ which reduce the remaining volume. These specific pyramid widths are denoted by w_{bottom} and w_{top}. Note that it is possible that $w_{bottom} = 0$ or $w_{top} = 0$, but it must always hold that

$$w_{bottom} + w_{top} = w_{max} - w_{update} \geq (1 - \gamma) w_{max}. \tag{11-2-9}$$

Now, what is the volume of the two new pyramids? In other words, what is the reduction of the remaining volume? This reduction of course depends on $w_{pyramid}$, but it also depends

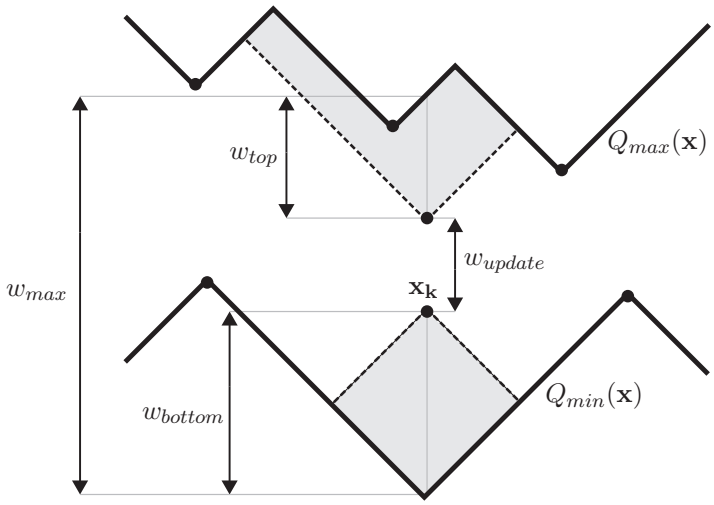

Figure 11-4: An example update and the corresponding change in volume.

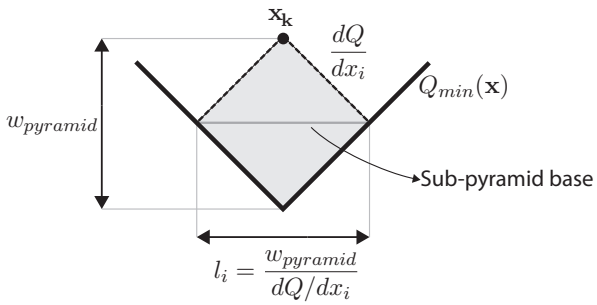

Figure 11-5: A visualization of a worst-case pyramid during an update, together with certain important parameters.

on the 'bottom surface area' onto which the pyramids fall, and this isn't generally known. It could be such as the top in figure 11-4, where the pyramid has a big volume because the pyramids above it slope away. On the other hand, it could also be such as the bottom in figure 11-4, where the pyramid has a relatively small volume. Which situation is the case cannot generally be established. However, the worst case scenario is the one shown at the bottom. This is the minimum pyramid volume that can ever occur for a given pyramid width $w_{pyramid}$.

The next question is: how much is this minimum reduction in volume? To find that out, consider only one side of such an update, as shown in figure 11-5. The reduction in volume now takes the shape of two sub-pyramids, connected at their base. Each of these sub-pyramids has a 'height' of $w_{pyramid}/2$. The base is a D-dimensional plane. Its length l_i in the direction of parameter x_i is given by

$$l_i = \frac{w_{pyramid}}{dQ/dx_i}. \tag{11-2-10}$$

The area of the sub-pyramid base thus is

$$A_{sub-pyramid} = \prod_{i=1}^{D} l_i = \prod_{i=1}^{D} \frac{w_{pyramid}}{dQ/dx_i} = \frac{w_{pyramid}^D}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-11}$$

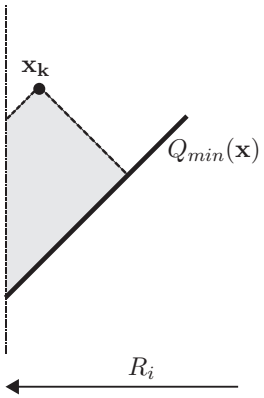

Figure 11-6: A special case occurs when an update is performed near the boundary of one of the parameters x_i.

The volume of a sub-pyramid thus becomes

$$V_{sub-pyramid} = \frac{A_{sub-pyramid}}{D+1}\frac{w_{pyramid}}{2} = \frac{1}{D+1}\frac{1}{2}\frac{w_{pyramid}^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-12}$$

However, it is important to keep in mind that, at the bottom of the update, there are two such sub-pyramids. The same holds for the top of the update. Hence, the reduction of volume ΔV (taken positive for volume reductions) is bounded by

$$\Delta V \geq \frac{1}{D+1}\frac{1}{\prod_{i=1}^{D} dQ/dx_i}\left(w_{bottom}^{D+1} + w_{top}^{D+1}\right). \tag{11-2-13}$$

Take a closer look at w_{bottom} and w_{top}. Minimizing the above quantity subject to constraint (11-2-9) results in

$$w_{bottom} = w_{top} = \frac{1-\gamma}{2} w_{max}. \tag{11-2-14}$$

That is, the volume reduction bound is smallest when $w_{bottom} = w_{top}$. This means that

$$\Delta V \geq \frac{2}{D+1}\left(\frac{1-\gamma}{2}\right)^{D+1}\frac{w_{max}^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-15}$$

This is a solid bound on the reduction of the volume.

The careful reader might have noticed a special case in which the above bound does not hold. And indeed, there is a hidden assumption in the above derivation. It has been assumed that the update pyramid does not cross the boundary of the parameter range R_i for any of the parameters x_i. However, this assumption doesn't always hold. It may occur that the update is close to the boundary of some parameter x_i. Such a case has been visualized in figure 11-6.

If this exception occurs, then the volume of the update pyramid is less than was previously expected. Luckily, $\mathbf{x_k}$ is always selected to fall within the allowed range. Thus, the pyramid volume can at most be halved in each direction. This results in the new bound

$$\Delta V \geq \frac{1}{2^D}\frac{2}{D+1}\left(\frac{1-\gamma}{2}\right)^{D+1}\frac{w_{max}^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-16}$$

Next, an even more careful reader might ask, 'What happens when a pyramid's volume is cut off on both sides, by both boundaries?' Indeed, in this case it may happen that a factor 1/2 is not sufficient. However, in reality this virtually never occurs. (It only happens when Q has been initialized with a width far larger than $w(R_i)dQ/dx_i$ for some parameter x_i. If Q_{min} and Q_{max} are adequately chosen, this will not happen.) And in the unlikely case that it does, it is again possible to derive a factor that circumvents the problem. But, due to the rarity of this issue, and to keep this explanation at least remotely resembling something simple and brief, the solution of this problem will be omitted.

The reader might also wonder what would happen when $\mathbf{x_{k+1}}$ does not fall within the range R_i for some parameter x_i. In this case, the function approximator cannot give a bound on $Q(\mathbf{x_{k+1}})$. However, it is assumed here that any state $\mathbf{x_{k+1}}$ that does not fall within the allowed range is a terminal state of the reinforcement learning problem. Its value is known exactly (i.e. with zero width), thus solving this issue.

11-2-3 Bounding the average width over time

Relation (11-2-16) gives a bound on the reduction of the volume during one iteration. This gives rise to the question how fast the volume is reduced over several iterations. Or, to be more exact, what is the average width $(w_{av})_k$ after k iterations?

First of all, from (11-2-3) it follows that the reduction of the average width during some iteration k is given by

$$\Delta w_{av} = \frac{\Delta V}{A} \geq \frac{1}{2^D} \frac{2}{D+1} \left(\frac{1-\gamma}{2} \right)^{D+1} \frac{1}{A} \frac{(w_{max})_k^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-17}$$

Since $\Delta w_{av} = (w_{av})_k - (w_{av})_{k+1}$, the ratio between two successive average widths is

$$\frac{(w_{av})_{k+1}}{(w_{av})_k} = 1 - \frac{\Delta w_{av}}{(w_{av})_k} \leq 1 - \frac{1}{(w_{av})_k} \frac{1}{2^D} \frac{2}{D+1} \left(\frac{1-\gamma}{2} \right)^{D+1} \frac{1}{A} \frac{(w_{max})_k^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-18}$$

Because $(w_{av})_k \leq (w_{max})_k$, it follows that

$$\frac{(w_{av})_{k+1}}{(w_{av})_k} \leq 1 - \frac{1}{2^D} \frac{2}{D+1} \left(\frac{1-\gamma}{2} \right)^{D+1} \frac{1}{A} \frac{(w_{av})_k^{D}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-19}$$

Now define the **reference width** w_r as

$$w_r = \left(\frac{D+1}{2} \left(\frac{2}{1-\gamma} \right)^{D+1} \prod_{i=1}^{D} w(R_i) \frac{dQ}{dx_i} \right)^{\frac{1}{D}}. \tag{11-2-20}$$

Note that this is a problem-specific constant which does not depend on k. This simplifies relation (11-2-19) to

$$\frac{(w_{av})_{k+1}}{(w_{av})_k} \leq 1 - \left(\frac{(w_{av})_k}{2w_r} \right)^{D}. \tag{11-2-21}$$

Define w_0 as the **initial width** $w_0 = (w_{av})_0 = Q_{max} - Q_{min}$. According to theorem 1 from appendix A (with $n = D$ and $x_d = 2w_r$), the average width is now bounded by

$$(w_{av})_k \leq \frac{2w_r}{\left(Dk + \left(\frac{2w_r}{w_0}\right)^D\right)^{\frac{1}{D}}}. \tag{11-2-22}$$

This not only proves that w_{av} converges to zero. It also gives a bound on the number of iterations required to obtain a certain accuracy.

11-2-4 Estimating the expected average width

It is nice to have a bound on the average width w_{av}, but this bound does not tell how quickly w_{av} is likely to converge to zero. It is only a very rough upper bound. To compensate for this, also an estimate can be made on the average change of w_{av}. It must be noted that an accurate estimate is impossible to make. The part below thus concerns a rough estimate.

First consider the constraint $w_{update} \leq \gamma w_{max}$. Previously, it was assumed that a worst case scenario occurred, and thus that $w_{update} = \gamma w_{max}$. In reality, this is not exactly the case. However, since during the algorithm execution the width of the Q-function does not vary much, it does approximately hold. It is thus once more assumed that $w_{update} \approx \gamma w_{max}$.

Secondly, it has previously been assumed that $w_{bottom} = w_{top}$. In reality, this is again not always the case. w_{bottom} and w_{top} do vary – especially during the start of the algorithm when the function approximator still needs to adjust to the main shape of the value function. However, while the function approximator is converging to the optimal value function, this approximation does start to become more accurate. Therefore, it is once more assumed that $w_{bottom} \approx w_{top}$.

Thirdly, it has been assumed that the update pyramid is projected on a 'valley-like' base, as shown in figure 11-5. This doesn't always hold. Sometimes it is indeed a 'valley-like' base, but sometimes it's also a 'tip-like' base, thus increasing the volume of the update pyramid. On average, it is expected that the update pyramid is projected on a roughly flat base. The volume of one update pyramid then becomes

$$V_{pyramid} = \frac{w_{pyramid}}{D+1} \prod_{i=1}^{D} \frac{2w_{pyramid}}{dQ/dx_i} = \frac{2^D}{D+1} \frac{w_{pyramid}^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-23}$$

Finally, there previously was a correction factor for the case where the update pyramid was near the state-action-space boundary. In reality, this situation rarely occurs, and on average such a correction factor thus does not need to be applied.

The above statements give an approximated change of volume

$$\Delta V \approx \frac{2^{D+1}}{D+1} \left(\frac{1-\gamma}{2}\right)^{D+1} \frac{w_{max}^{D+1}}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-24}$$

This in turn results in a recursive relation for the average width, being

$$\frac{(w_{av})_{k+1}}{(w_{av})_k} \approx 1 - \frac{2^{D+1}}{D+1} \left(\frac{1-\gamma}{2}\right)^{D+1} \frac{1}{A} \frac{(w_{av})_k^D}{\prod_{i=1}^{D} dQ/dx_i}. \tag{11-2-25}$$

Using the definition for the reference width, this can also be written as

$$\frac{(w_{av})_{k+1}}{(w_{av})_k} \approx 1 - \left(\frac{2(w_{av})_k}{w_r} \right)^D . \tag{11-2-26}$$

The average width is thus approximated as

$$(w_{av})_k \approx \frac{w_r}{2\left(Dk + \left(\frac{w_r}{2w_0} \right)^D \right)^{\frac{1}{D}}} . \tag{11-2-27}$$

This quantity indeed decreases much faster than the bound given earlier. The question remains whether it is accurate. To find that out, the next chapter will assess the accuracy of these bounds through several practical examples.

Chapter 12

Practical implementations of the continuous Q-learning algorithm

The previous chapter has introduced the continuous interval Q-learning algorithm. This chapter will apply this algorithm (or at least, various versions of it) to several simple test systems. First a direct application of the algorithm will be set up. This algorithm has some significant downsides, so afterwards an alternative algorithm will be designed that eliminates these downsides. In the end, this alternative algorithm is applied to more complicated systems, including the CAP system. The chapter closes off with some suggestions for further research.

12-1 A direct implementation of the algorithm

This section will try to implement the continuous Q-learning algorithm as it has been set up in the previous chapter. First some issues in its implementation will be discussed, after which the algorithm will be implemented and results will be shown.

12-1-1 Issues with the continuous Q-learning algorithm

During algorithm 1, four things need to be done with the Q-function approximator. It needs to be possible to find values $Q(\mathbf{x})$, the lower bound $\underline{Q}$ needs to be maximized, the upper bound $\overline{Q}$ needs to be maximized and the width $w(Q)$ needs to be maximized. But is all of this easily achieved?

First consider looking up a value of Q. This is what algorithm 2 does. Computationally, how fast does it do this? It is easily shown that this is of order $\mathcal{O}(n_{iterations})$. (This notation, roughly speaking, means that the run-time is proportional to the number of iterations $n_{iterations}$.) All previous updates simply need to be examined to find the value Q. Though this order is not exactly ideal – most function approximators allow look-ups in constant ($\mathcal{O}(1)$) time – it is still acceptable.

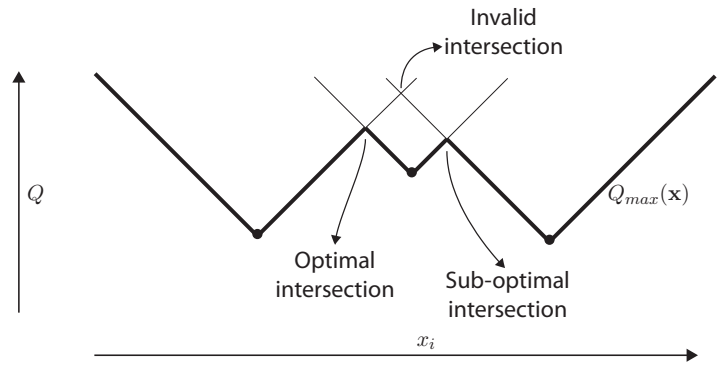

Figure 12-1: The process of maximizing Q_{max} requires looking at intersections of pyramids. Some intersections are invalid and should be ignored. Of the valid intersections, the highest one should be returned as the maximum of Q_{max}.

Now consider the problem of maximizing the lower bound $\underline{Q}$. This maximum must always lie at the top of one of the update pyramids. To maximize the lower bound, it is thus only required to check the tops of all the update pyramids. This can also be done in $\mathcal{O}\left(n_{iterations}\right)$ time, which is still acceptable.

The problems arrive when one tries to maximize $\overline{Q}$. To see why, first consider the one-dimensional case as shown in figure 12-1. The maximum of $\overline{Q}$ is always at the intersection of two update pyramids. Thus, to find it, all pairs of update pyramids need to be considered. For every pair, the intersection needs to be found. It then needs to be checked whether this is indeed a valid point on $Q(\mathbf{x})$. Of all valid intersections the highest one then needs to be returned. This algorithm is of $\mathcal{O}\left(n^2\right)$ (where, for brevity, $n_{iterations}$ has been shortened to n).

A reader with some knowledge of algorithms would realize here that it's also possible to design a clever data structure to store the update pyramids, thus bringing the run-time down to $\mathcal{O}\left(n\log(n)\right)$. This is indeed true. However, now consider the multi-dimensional case. In this case, comparing every pair of update pyramids is no longer sufficient. Instead, one should compare every combination of $D+1$ update pyramids. The run-time of the algorithm will thus be something like $\mathcal{O}\left(n^D\log(n)\right)$. For $D=2$ this already poses practical problems, and for problems with $D>2$ this is simply unacceptable.

It might be possible to reduce the run-time somewaht by eliminating old and useless update pyramids. If an update pyramid is fully enclosed by another pyramid, it has become useless and can be erased from the data storage. Practical tests have shown that this will cut the number of active update pyramids roughly in half. Though this is definitely useful from a practical point of view, it does not change the order of the algorithm, and thus the problem is not solved. The algorithm is therefore not suitable for any practical implementation.

12-1-2 A first application of the algorithm

Still, it would be nice to at least see if the continuous interval Q-learning algorithm works. Therefore, it has been applied to a simple two-dimensional problem.

Consider the simple problem where the state s describes some position x and the action a is the velocity $\dot{x}$. Both the position s and the velocity a fall within the interval $[-1,1]$. The time

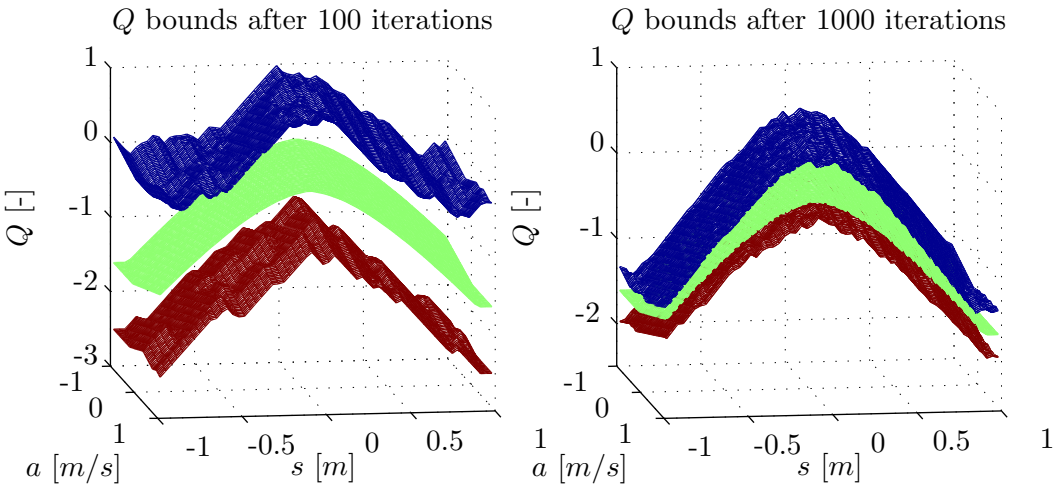

Figure 12-2: Plots of the bounds on Q after 100 (left) or 1000 (right) runs of the continuous interval Q-learning algorithm. The green surface area is the optimal value function Q^*, which is bounded by the upper (blue) and lower (red) bound.

step used is $\Delta t = 0.1$ s. (The controller can basically choose Δs from the interval $[-0.1, 0.1]$.) The reward given at every time step is $r_k = |s_k|$. (The agent needs to move the cart to $s = 0$.) Furthermore, $\gamma = 0.8$, $Q_{min} = -5$, $Q_{max} = 2$, $dQ/ds \leq 5$ and $dQ/da \leq 0.5$. Since this problem has $D = 2$, it is referred to as the **simple two-dimensional problem**.

The continuous interval Q-learning algorithm has been applied to this problem. A test has been run both for $n_{iterations} = 100$ and $n_{iterations} = 1000$. While the first case had an acceptable run-time (a few seconds), the second case already required significant waiting (nearly an hour on a reasonably fast home computer). The results are shown in figure 12-2.

In the figure, it can be seen that the bounds converge to the optimal value function Q^*, as was expected. An attentive reader might note that the lower bound (red) converges slightly faster than the upper bound (blue). This can be explained. Due to the pyramid-like nature of the bounds, there are always peaks in the bounds. That is, there are slightly higher values and slightly lower values. When updating the bounds, one tries to maximize $\underline{Q}$ and $\overline{Q}$. Because of these peaks, both of these values will be slightly higher than expected. When updating the lower bound, this is positive. The lower bound goes up a bit faster. When updating the upper bound, however, this is not positive. The upper bound stays high and is narrowed somewhat slower than otherwise would be the case. Hence, these peaks are benificial for the lower bound but not beneficial for the upper bound. This is why the lower bound converges slightly faster than the upper bound.

Another interesting thing to investigate is the width of Q as the algorithm progresses. The minimum width, the maximum width, the average width, the average width bound according to relation (11-2-22) and the expected average width according to relation (11-2-27) are all shown in figure 12-3.

First of all, it can be seen that the bound is not violated. But, as expected, it is much too rough. The actual average width (and even the maximum width) is far below it. On the other hand, the expected average width does provide a reasonable estimate of the actual average width. As the iteration number becomes big, there is a bit of an offset, but this offset is not all that big. Hence the average expected width relation will be a useful tool to estimate how

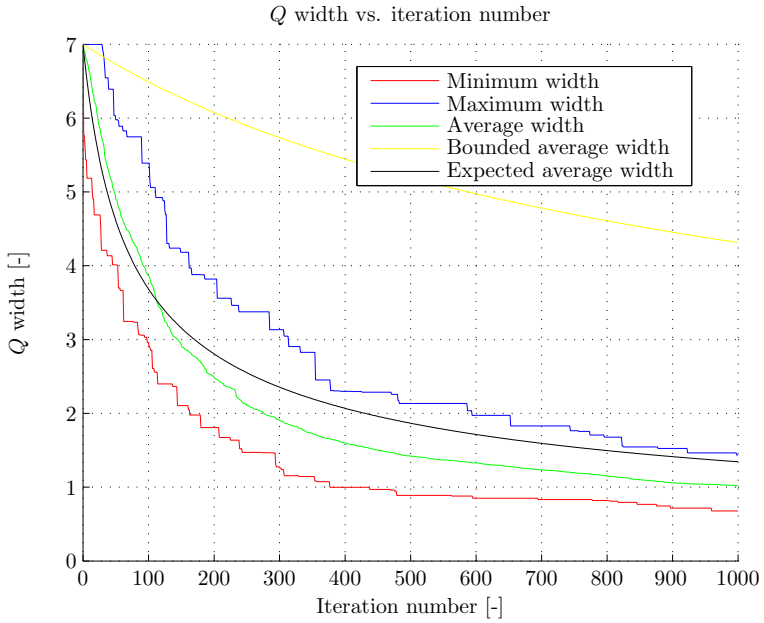

Figure 12-3: Evolution of the width of Q during a run of the continuous interval Q-learning algorithm.

many iterations it will take for the algorithm to reach a certain accuracy.

12-2 Splitting the state-action-space up into set blocks

The previous section already showed an algorithm which worked, but it was so slow that it was not practically feasible. Finding Q and especially maximizing $\overline{Q}$ simply took too long. This section considers a first attempt at solving this problem. First the main idea behind the solution is explained. Then the new algorithm is analysed, both from a theoretical point of view and through a practical example.

12-2-1 The main idea

The main idea is to split the state-space up into blocks. And for this first version, the blocks all have the same constant size. Each block has a corresponding (interval) Q-value. To then find a Q-value for some point $\mathbf{x}$, the Q-function approximator simply looks at which block the point $\mathbf{x}$ is in and returns the value of the corresponding block.

The important part of this algorithm is the updating of the blocks. Such an update is shown in figure 12-4. It is very important that the bounds remain consistent and can never be violated. It may not occur that the actual Q^* value falls outside of the Q interval. Thus, when performing an update, the blocks can only be narrowed so much that they do not cross the update pyramid corresponding to the update.

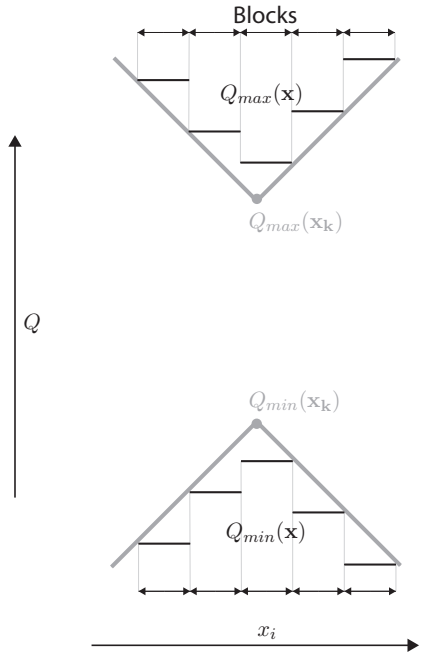

Figure 12-4: The process of updating block values, such that the bounds remain consistent. The blocks can be narrowed until they hit the update pyramid. Any further narrowing could result in an incorrect bound.

12-2-2 Analysing the new algorithm

The algorithm is a significant improvement from its precursor. Q-values can be looked up in $\mathcal{O}(1)$ time. Maximizing a Q-value bound does still require some time – all blocks need to be checked – but this is a time which does not increase when $n_{iterations}$ increases.

There are only two significant downsides. First of all, the algorithm will converge a bit slower than its predecessor. This is because not all data is optimally used. There are still 'holes' in the update pyramid which are not filled. However, the effect of this disadvantage remains limited.

The most important disadvantage is the inability of this algorithm to accurately adjust to the actual Q-function as the algorithm progresses. The number of blocks remains constant during run-time, and hence after a while the algorithm simply cannot improve any further. This is a significant problem that needs to be solved.

12-2-3 A practical application

The constant block algorithm has been applied to the simple two-dimensional problem. This has been done for two different block sizes. Either a total of 400 blocks (20 in each direction) has been applied, or 40.000 blocks (200 in each direction), in which the latter case of course had a longer run-time. The results can be seen in figure 12-5.

First of all, it can be noticed that, if the blocks are too big, then the algorithm doesn't converge properly. It continues to be somewhat distant from the actual Q^* value, even after ten million

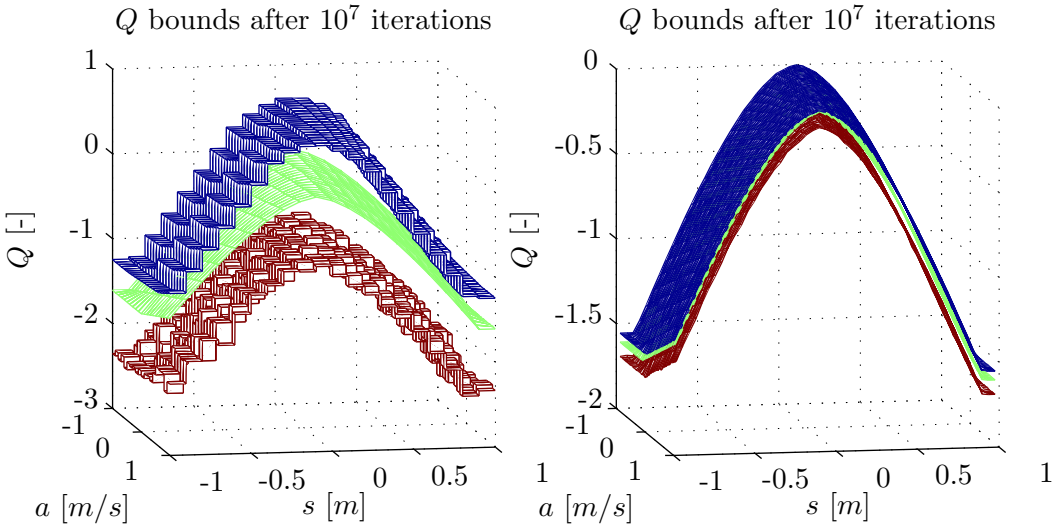

Figure 12-5: Plots of the bounds on Q after 10^7 iterations of the continuous interval Q-learning algorithm with constant size blocks. The left graph has been trained with big blocks (20 per direction), while the right graph has been trained with small blocks (200 per direction).

iterations. When the blocks are small enough, the algorithm does converge properly. The Q^* function is bounded quite accurately.

Of course the big question here is: how does one know how many blocks are required? This is not known in advance, and it's also not an easy question to answer. Using too little blocks doesn't work, while using too many blocks could result in too long run-times. A more appropriate solution would be to vary the number of blocks during run-time. And that is indeed the improvement that will be explored in the upcoming section.

12-3 Dynamically varying the number of blocks

This section will introduce another extension to the algorithm: dynamically splitting up blocks. At some point in time, there is a block and it needs to be split up into several sub-blocks. There are two obvious questions: how is this done and when is this done? These two questions are answered first. Then the resulting algorithm is again analysed.

12-3-1 How to split up blocks

Suppose that a block will be split up into sub-blocks. How will it be split up? Or, to be more precise, how many sub-blocks will be made, and where will the boundaries of these sub-blocks be?

One option would be to conveniently choose the boundaries of sub-blocks, such that updates are performed more smoothly. This seems like a promising idea, but in reality it's a lot less promising. Suppose that at the start of the algorithm run a block is split up at a certain splitting point. As the algorithm progresses, more and more update pyramids are added and stacked up on each other. After a few additional iterations, the splitting point that was so conveniently placed earlier on has lost its reason for being positioned where it is. The

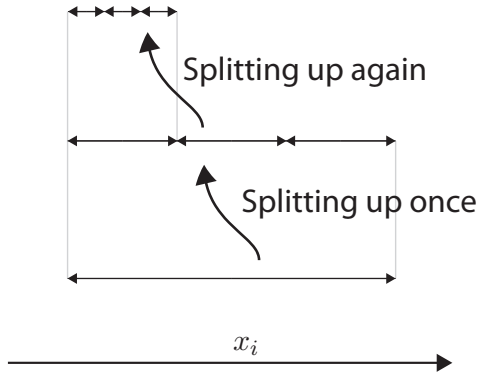

Figure 12-6: Splitting up a block into sub-blocks is done as shown above. Blocks are split up uniformly in a set amount of (in this case 3) sub-blocks per direction. Sub-blocks can then later on, if required, again be split up. This creates a tree-like structure of blocks.

usefulness of this idea is thus strongly limited. It might be possible to get this idea to work, but that will be complicated and is thus merely posed as a suggestion for further research in section 12-7.

The solution here is to go for a simple yet effective approach. When a block is split up, it is split up uniformly into sub-blocks. At the start of the algorithm, the user specifies manually into how many sub-blocks the blocks are split up, for each dimension. (For example, it might be specified that sub-blocks are split up into three blocks in the direction of s, but only in two blocks in the direction of a. This could be useful if the state s benefits more from a high resolution than the action a.) Whenever a block then needs to be split up into sub-blocks, it is then uniformly split into that amount of sub-blocks. This process is shown in figure 12-6.

12-3-2 When to split up blocks

The second question to be answered is when to split up sub-blocks. Suppose that an update is being performed for some point $\mathbf{x_k}$. If a given block can be narrowed, thanks to this update, then there appears to be no problem whatsoever. There is no need to split up the block further. This would only cost more computational time with hardly any gain. Improvements are already being made.

But now suppose that the block cannot be narrowed. Then the obvious question becomes: if the block was split up into sub-blocks, could some of these sub-blocks be narrowed? This idea is shown in figure 12-7. If a split of the block would result in improvements, then it would make sense to split up the block.

The only problem is that such an algorithm set-up would result in an infinite loop. To see why, consider a block which is passed through by the update pyramid (like the left-most sub-block of figure 12-7). If this block is split up, then some of the resulting sub-blocks might be narrowed, which is an improvement. This could be seen as an argument to split up the block. However, the result is that there will be another sub-block which the update pyramid passes through. The process thus continues, and blocks will continue to be split up.

This problem can be solved by adding another split-up criterion. A block may only be split up if the block actually contains the update point $\mathbf{x_k}$. This ensures that only blocks are split

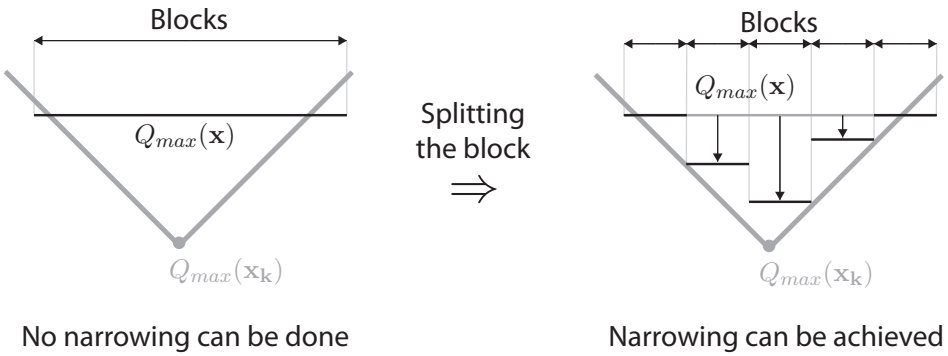

Figure 12-7: A case where it's useful to split up a block. Before the split, the block value could not be narrowed. After the split into 5 sub-blocks, some of the sub-blocks' values could be narrowed, thus reducing the remaining volume.

up that are actually visited. Furthermore, points that often get visited automatically get more sub-blocks and thus more resolution in this way. This solution thus also makes sense intuitively.

12-3-3 Analysing the algorithm

So how good is this adjusted algorithm? The expectation is that it runs slower than the previous constant-block-size algorithm, but still much faster than the original continuous interval Q-learning algorithm. However, the situation is opposite when the efficiency with which the algorithm uses data is considered. The original algorithm uses all data as efficiently as possible, whereas the other two algorithms regularly 'waste' data. That is, during an update they don't 'fill up' an entire update pyramid. For this reason, it is expected that the original algorithm narrows faster, given the same amount of iterations.

These expectations are indeed confirmed by a practical example. This time 10^4 simulations have been run on the simple two-dimensional system. Doing so only cost a few seconds, as opposed to the few hours that had to be waited for only 10^3 iterations of the original algorithm. Results of this test are shown in figure 12-8. Here it can be seen that the algorithm does still converge. However, it converges a bit slower than what was previously the case with the original algorithm. That is, the green line (indicating the actual average width) is a bit above the black line (indicating the expected average width), while previously the real average width was quite a bit below the expected average width.

Although good progress has been made so far, the algorithm should still be improved. The performance needs to increase. The obvious way to do that is to make sure that more parts of the update pyramid are used in actual updates. Currently, because flat blocks are used, a lot of information is wasted. A way in which this can largely be prevented, is by allowing blocks to not be flat. Instead, they can be sloped. That will be the subject of the next section.

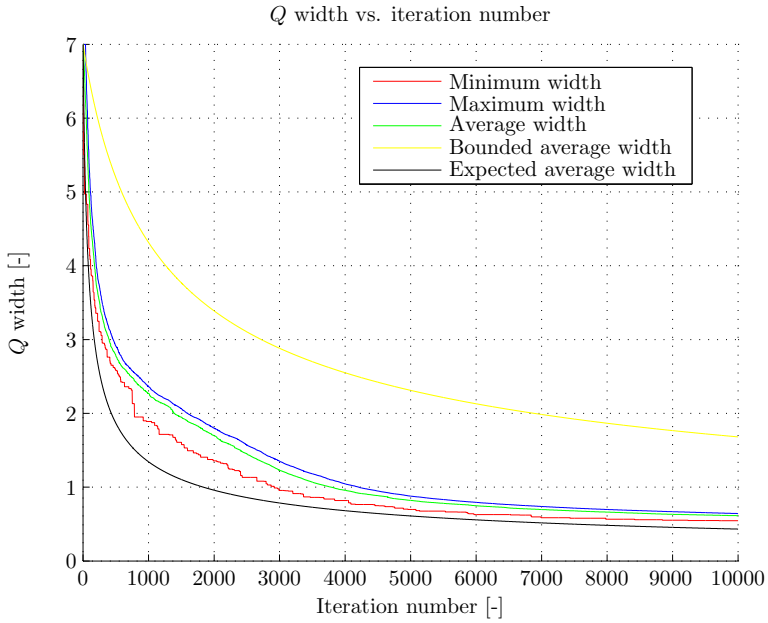

Figure 12-8: Plots of the bounds on Q after 10^4 iterations of the continuous interval Q-learning algorithm with flat blocks of varying size.

12-4 Allowing sloped blocks

Suppose that blocks don't have to be flat. Suppose that they can be sloped lines (for one dimension) or planes (for multiple dimensions). How can this be implemented? And what will the result be? That's what this section will look at.

12-4-1 Implementation of the extension

First of all, to allow sloped blocks, the representation of a block needs to change. One now doesn't only need to keep track of the interval Q-value. One also needs to know the slopes of the minimum and maximum of the block in each direction. Keeping track of this isn't all that hard though.

A somewhat harder problem is deciding when and how to update the blocks. It is important here to keep in mind that the goal of the algorithm is to minimize the remaining volume. The remaining volume of any particular block equals the block area (which is constant) multiplied by the average width of the block. This average width equals the width at the midpoint of the block. Thus, if the width of the midpoint can be decreased, then the block should be updated. This idea is visualised in figure 12-9. Here it can be seen that blocks are updated if it means that the interval at their midpoints will narrow. This is not the case for the block on the right, and hence it is not updated.

There are still two slight downsides to this update method. First of all, still not the entire update pyramid is used during an update. Generally, the tip of the pyramid itself is not used, as can be seen in figure 12-9. However, a much bigger part of the update pyramid is used than previously, so this small downside can be accepted.

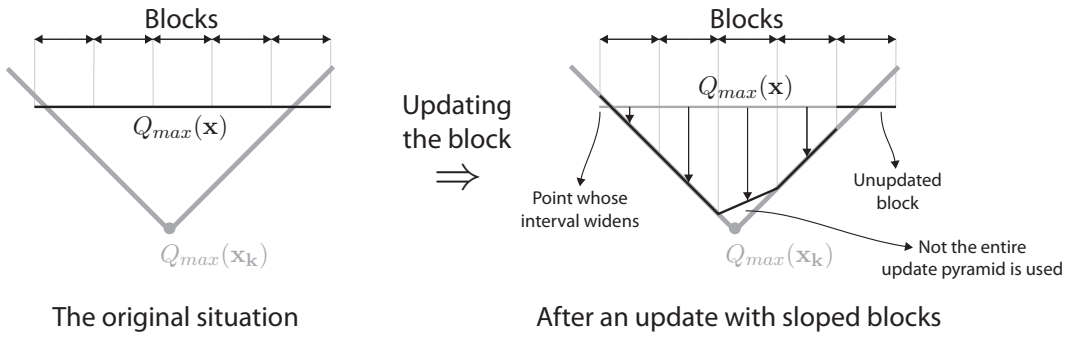

Figure 12-9: Updating of blocks with slopes. This increases performance, though there are still a few downsides.

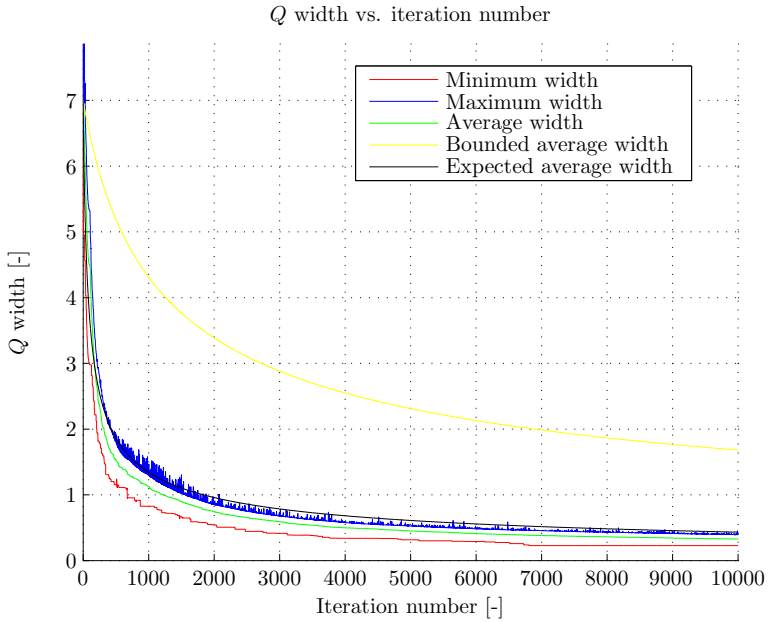

Figure 12-10: Plots of the bounds on Q after 10^4 iterations of the continuous interval Q-learning algorithm with sloped blocks of varying size.

The second downside is that sometimes there are points whose interval values actually widen. This occurs at the left sub-block of figure 12-9. It seems strange: why would one widen the interval value of a point? However, this is the way in which the maximum reduction of the remaining volume can be obtained. That is why it occasionally can occur that, for some specific point $\mathbf{x}$, the width $w(Q(\mathbf{x}))$ actually increases. However, it is important to keep in mind that the remaining volume V and the average width w_{av} always do decrease.

12-4-2 Analysing the extension

Now that sloped blocks are allowed, what happens to the performance of the algorithm? Does it perform better than when flat blocks were used? To find that out, the same test has been run as the one discussed in the previous section. The results are shown in figure 12-10.

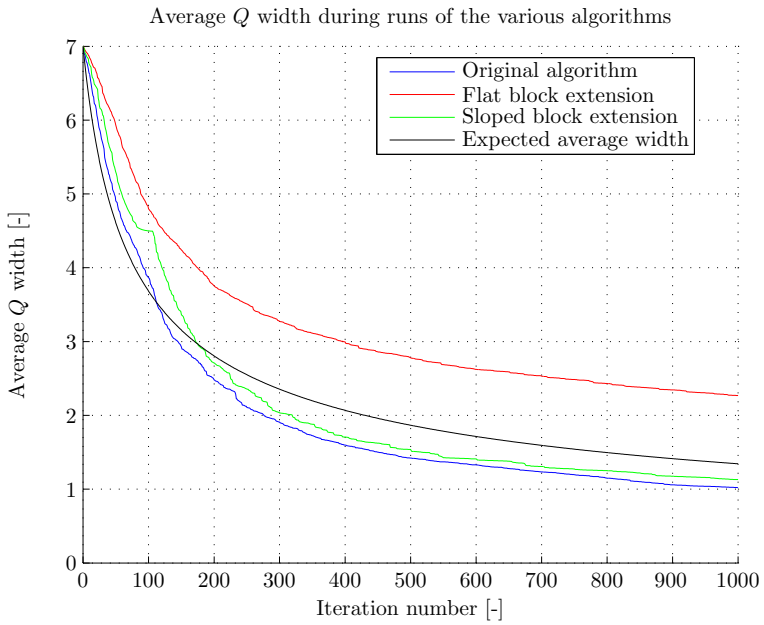

Figure 12-11: Comparison of the various algorithms so far. The average width has been plotted versus the iteration number.

In figure 12-10 it can be seen that the algorithm with sloped blocks performs significantly better than the case where flat blocks were still used. The average width again falls well below the expected average width. Another interesting thing that can be noticed is that the maximum width (the blue line) initially increase to above 7 (which is its initial value). Only after roughly 30 iterations does it drop below 7 again. This is indeed due to the fact that the width of a particular point can increase, as was just discussed.

To really compare the effect of the sloped blocks, a comparison plot has been made. Figure 12-11 shows the average width of three algorithms: the original algorithm, the algorithm with a varying number of flat blocks and the same algorithm with sloped blocks. It now becomes evident that using flat blocks doesn't work all that well. On the other hand, using sloped blocks does. Of course the sloped blocks algorithm performs slightly worse than the original algorithm – it's theoretically very improbable for it to perform better – but the difference is nearly negligible. And since the run-time is significantly faster, it is indeed a useful improvement of the algorithm.

12-5 Detailed workings of the varying block algorithm

In the previous sections, a short indication has been given of how the varying block algorithm works. That is, when and how blocks are split up and how they are updated. However, it is also necessary to look up Q interval values, as well as maximizing $\underline{Q}$ and $\overline{Q}$ values. How is that done in detail? That is discussed in this section. This section then finishes off with a detailed analysis of the varying-block algorithm, including a proof that it also converges to the optimal value function Q^*.

12-5-1 Looking up a Q value

First of all, consider the problem of looking up a Q value. How is this done? The process is actually quite simple. As was already explained with figure 12-6, the whole process of splitting up blocks creates some kind of tree structure. To find a Q value, one simply needs to follow the tree, along the branches, to the right leaf.

This cryptic explanation might be a bit too brief for some readers, so things will be further elaborated now. At the start of the algorithm run, there is only one block, called the **main block**. This block spans the entire state-action-space. When this block is split up, it gets a certain amount of sub-blocks. When each of these sub-blocks is then split up, it also gains sub-blocks of its own. However, it is important to keep in mind that, when splitting up a block, the 'block' as an object does not disappear. It only changes from a **leaf** – something that keeps track of Q values – to a **node** – something that branches out to other nodes/leafs.

Now let's say that one wants to find the Q value of a certain point $\mathbf{x}$. The search for this value starts at the main block. The main block checks which of its sub-blocks contains the point $\mathbf{x}$. (Such a sub-block must exist. After all, the main block must contain $\mathbf{x}$ – it contains all points – and all sub-blocks of the main block cover the entire main block.) It then passes the request of finding $Q(\mathbf{x})$ on to the corresponding sub-block. This block then in turn checks which of its sub-blocks contains $\mathbf{x}$. Only when the algorithm arrives at a leaf – a block without sub-blocks – will the leaf calculate and return the actual value $Q(\mathbf{x})$. This process is summarized in the following algorithm, which is initially called using the main block.

Algorithm 3 – The varying-block Q function – Finding Q

Require: A block, and a point $\mathbf{x}$ in that block whose value needs to be found.
  if the block has sub-blocks **then**
    Find the sub-block that contains $\mathbf{x}$.
    Ask this sub-block for the value $Q(\mathbf{x})$ and return it.
  else
    Calculate the value $Q(\mathbf{x})$ and return it.
  end if

The next question to be answered is: what will the order of this process be? Looking up in which sub-block a certain point is located can be done in $\mathcal{O}(D)$ time. But, assuming that D is constant anyway, this can also be seen as $\mathcal{O}(1)$. The only repetitive task then is the process of climbing up the tree. If there are n_{blocks} in the algorithm, then this climb will take $\mathcal{O}(\log(n_{blocks}))$ time. Thus, looking up a Q-value also takes $\mathcal{O}(\log(n_{blocks}))$ time.

12-5-2 Maximizing $\underline{Q}$, $\overline{Q}$ and $w(Q)$

It also often occurs that certain quantities need to be maximized. Examples are $\underline{Q}$, $\overline{Q}$ and $w(Q)$. Sometimes these quantities need to be maximized globally. For example, in the best exploration strategy, the point $\mathbf{s}, \mathbf{a}$ which maximizes $w(Q(\mathbf{s}, \mathbf{a}))$ needs to be found. Sometimes though, these quantities need to be maximized subject to certain constraints. For example, one often needs to find the action $\mathbf{a}$ which maximizes $\overline{Q}(\mathbf{s}, \mathbf{a})$ for a given state vector $\mathbf{s}$. How can this maximization best be done? (For simplicity of notation, it will be assumed in this subsection that one wants to maximize $\overline{Q}(\mathbf{x})$. Maximization of other quantities is done in a similar way.)

One suggestion might be to simply look everywhere, in all sub-blocks, for the highest value of $\overline{Q}(\mathbf{x})$. Doing so will require a lot of time though. One has to examine almost n_{blocks} blocks, which is computationally not very efficient. This option is thus quickly rejected.

Another suggestion might be to keep track of the global optimum of $\overline{Q}(\mathbf{x})$. This is an interesting suggestion. For $\underline{Q}(\mathbf{x})$, this suggestion might work, as $\underline{Q}(x)$ is a quantity which only increases. (Okay, with one minor exception as indicated earlier, but that can be worked around.) It is thus possible to keep track of some global maximum $\underline{Q}_{max}$. Every time an update occurs for some block, it is checked whether this update creates a new global maximum $\underline{Q}_{max}$ within that block. If so, a new global maximum is stored.

The reason why this doesn't work for $\overline{Q}(\mathbf{x})$ is that $\overline{Q}$ is a quantity which can decrease. (In fact, it almost only decreases.) One can keep track of the global maximum, of course, but when this maximum is updated, one needs to search the entire state-action-space for a new global maximum. This is rather cumbersome. Furthermore, knowing the global maximum $\overline{Q}_{max}$ wouldn't even be sufficient, since $\overline{Q}$ doesn't need to be maximized globally. Instead, it needs to be maximized for an action $\mathbf{a}$, given a certain state $\mathbf{s}$. Thus, global maximums need to be kept track of for every state $\mathbf{s}$. In theory this is possible, but in practice this is rather inefficient.

The solution here lies in again cleverly using the tree structure that is present. Every block needs to keep track of which possible values of $\overline{Q}$ can be present within all of its sub-blocks. (These values are stored in an interval.) Then, given that data, one can browse through the tree structure, ignoring (pruning) parts that are certain not to contain the maximum. This is done according to the following algorithm.

Algorithm 4 – The block Q function – Maximizing $\overline{Q}$

Initialize the maximum m as Q_{min}.
Create a stack of blocks.
Put the main block on the stack.
while there are blocks on the stack **do**
  Get the top block off the stack.
  if all possible values of $\overline{Q}$ in the block are below m **then**
    Completely ignore this block. Continue with the next block in the stack.
  end if
  if the block has sub-blocks **then**
    Add all the sub-blocks (satisfying the criteria) to the stack.
  else
    The block must have a point with $\overline{Q} > m$, or it would've already been ignored.
    Update the maximum m to be the maximum possible value of $\overline{Q}$ within the block.
    Remember at which point this maximum occurs.
  end if
end while
Return the last remembered point.

This algorithm basically considers all block which might potentially contain the point that maximizes $\overline{Q}$. If a block cannot contain such a point, then this block and all its sub-blocks are ignored. In this way, a big part of the search space is ignored, thus resulting in a reasonably efficient algorithm.

So how efficient is this algorithm? That strongly depends on the Q function itself. In a worst-case scenario, all leafs have approximately the same value. In that case, the algorithm has to consider all leafs to find the point that maximizes $\overline{Q}$. The algorithm would be of order $\mathcal{O}(n_{blocks})$. If on the other hand there are some strong variations in $\overline{Q}$, then the search will be more efficient. An order of $\mathcal{O}(\log(n_{blocks}))$ or $\mathcal{O}\left(\log(n_{blocks})^2\right)$ or so can be achieved.

When maximizing $\underline{Q}$ or $\overline{Q}$ this latter scenario occurs. Due to variations in the values, the search algorithm is very efficient. However, when maximizing $w(Q)$, this is most certainly not the case. Due to the set-up of the algorithm, all points $\mathbf{x}$ have similar widths $w(Q(\mathbf{x}))$. As a result, almost all blocks are considered in the search to maximize $w(Q)$. This is a significant factor in slowing down the algorithm. It causes this algorithm to have run-time $\mathcal{O}(n_{blocks})$.

Do keep in mind that, for this algorithm to work, it needs to keep track of the possible values of $\overline{Q}$, $\underline{Q}$ and $w(Q)$ that are present within it. How exactly it does this is something that will (among others) be discussed in the upcoming section.

12-5-3 Updating the Q-function

Previously, the whole process of updating the Q-function has already been discussed. It just hadn't been summarized in an algorithm yet, because some parts were missing. Most specifically, it wasn't certain yet how to determine which blocks required updating. But this is known now. One simply needs to walk through the tree in a clever way again. This trick is worked out in the algorithm below, which is initially called with the main block as argument.

Algorithm 5 – The block Q function – Updating a block

Require: A point $\mathbf{x_k}$ of which a new interval $Q(\mathbf{x_k})$ is known.
Require: A block to be updated.
if no part of the block can benefit from an update whatsoever **then**
    Ignore the whole block including its sub-blocks. Return immediately.
end if
if the block does not have sub-blocks **then**
    if the block can be updated immediately **then**
        Narrow the bounds of the block as much as possible towards the update pyramid.
        Pass information of the new bounds down to parent blocks.
    else
        if the block might partly benefit from an update **then**
            if the block contains $\mathbf{x_k}$ **then**
                Split the block up into sub-blocks. Continue with the part below.
            else
                Ignore the block. Return immediately.
            end if
        else
            Ignore the block. Return immediately.
        end if
    end if
end if
When this point in the algorithm is reached, the block must have sub-blocks.
if the block might partly benefit from an update **then**

Individually update all sub-blocks.
else
Ignore this block and all sub-blocks.
end if

So how efficient is this updating? Does it really consider only a small part of all the blocks? That question has been researched with the simple two-dimensional problem. It turns out that for the first 10 iterations or so, nearly 100% of the blocks are considered for updating. After 100 iterations, that's down to $50-60\%$, after 1.000 iterations it's $20-25\%$ and after 10.000 iterations it's $4-8\%$. That raises the question: how many blocks are updated during an update?

A decent rule of thumb (ignoring parts of the function approximator with disproportionally high or low numbers of blocks) is that the number of blocks that are updated is given by

$$n_{blocks\,updated} \approx \frac{A_{pyramid}}{A_{total}} n_{blocks}. \tag{12-5-1}$$

It is known that A_{total} is constant. Also, $A_{pyramid}$ is proportional to w_{av}^D. As the iteration number k grows, relation (11-2-22) shows that w_{av} decreases proportionally to $k^{-1/D}$. Hence, for large iteration numbers, $A_{pyramid}$ is proportional to $1/k$. It can also be shown that n_{blocks} is roughly proportional to the iteration number k. This implies that $n_{blocks\,updated}$ converges to a constant value as $k \to \infty$. Whether this is actually supported by experiments is not fully clear yet. What is clear, is that $n_{blocks\,updated}$ is not of $\mathcal{O}(n_{blocks})$ but of either $\mathcal{O}(1)$ or $\mathcal{O}(\log(n_{blocks}))$.

There is still one line in the above algorithm that deserves some extra attention. It's the line 'Pass information of the new bounds down to parent blocks.' The algorithm to maximize $\overline{Q}$ (algorithm 4) requires blocks to know the range of $\overline{Q}$ which its sub-blocks have. Whenever a sub-block is updated, a check should be performed whether this data also requires updating. So generally the line means that data on the sub-block is passed down to parents such that they can keep track of their children.

Taking into account this fact, what's the order of the algorithm? The process of passing on information to parents is of order $\mathcal{O}(\log(n_{blocks}))$. The number of blocks that are updated themselves converges to some constant value, so it's $\mathcal{O}(1)$. (Though in practice, for small to medium values of the iteration number k, the number of updated blocks behaves more like $\mathcal{O}(\log(n_{blocks}))$.) The updating of a sub-block itself costs $\mathcal{O}(1)$ time as well. Hence the order of the entire update algorithm is either $\mathcal{O}(\log(n_{blocks}))$ or $\mathcal{O}\left(\log(n_{blocks})^2\right)$. Both options are well below $\mathcal{O}(n_{blocks})$ and are therefore definitely satisfactory.

12-5-4 General algorithm analysis

Now that all the algorithms have been set up, it is time to investigate the order of the entire combined algorithm. This investigation starts at algorithm 1. This algorithm has a loop which executes algorithm 4 (three times) and algorithm 5. It may also execute algorithm 3 a certain number of times, depending on the exact implementation of the algorithm.

The main bottleneck in this algorithm is algorithm 4, applied to maximize $w(Q)$. This has run-time $\mathcal{O}(n_{blocks})$. Compared to that, all the other algorithms (with run-time $\mathcal{O}(\log(n_{blocks}))$) are irrelevant. A single execution of the loop in algorithm 1 thus costs $\mathcal{O}(n_{blocks})$ time.

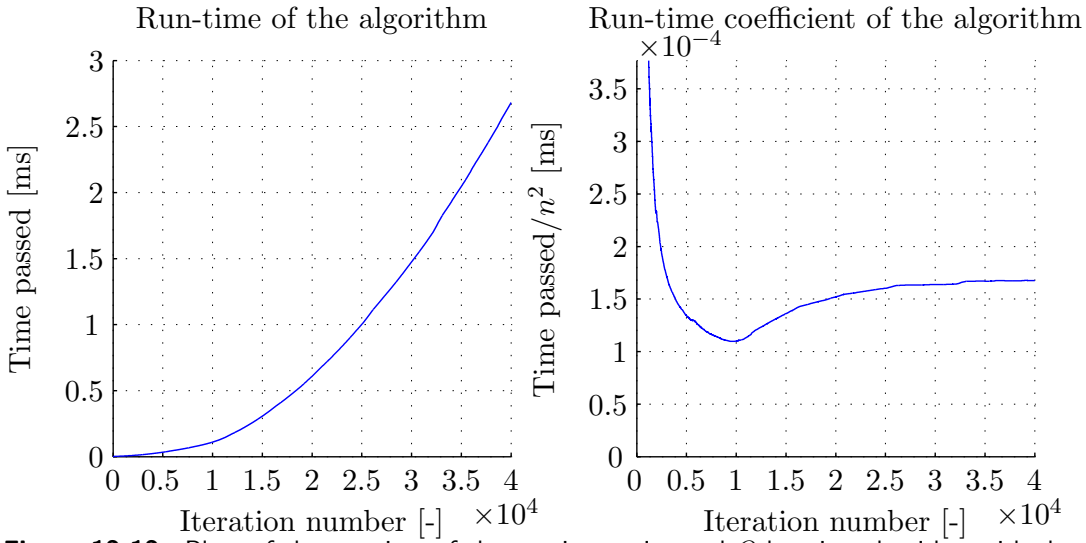

Figure 12-12: Plots of the run-time of the continuous interval Q-learning algorithm with the best exploration strategy. On the left is the run-time versus the iteration number. On the right is the run-time divided by k^2 versus the iteration number. It converges to a constant value.

To find the order of the entire algorithm, the relationship between the number of blocks n_{blocks} and the iteration number k should be known. It can be shown that these two are proportional. After all, every time a block is split up, n_{blocks} increases with a constant number. And only during a certain percentage of the updates (in practice roughly 20%) will a block split up. Thus n_{blocks} increases roughly linearly with the iteration number k.

Now the order of algorithm 1 can be obtained. A number $n_{iterations}$ times will a loop of order $\mathcal{O}(n_{blocks})$ be executed, where n_{blocks} is proportional to k. This implies that the algorithm is of order $\mathcal{O}\left(n_{iterations}^2\right)$.

To see whether this is true in practice as well, the run-time of the actual algorithm after k iterations should be considered. Such a plot is shown in figure 12-12 on the left. It can already be seen that the graph shape resembles the shape of a parabola. However, to be certain whether the determined order of the algorithm is accurate, the run-time at any time k should be divided by k^2. If the run-time divided by k^2 converges to a more or less constant value, then the order of the algorithm appears to be correct. And, as the right part of figure 12-12 shows, this is indeed the case. The run-time of the algorithm can (for the specific computer that used for these experiments) thus be approximated by $t_{run} \approx 1.7 \cdot 10^{-4} \cdot n_{iterations}^2$.

More interesting things can be determined though. Previously, it was found that the bottleneck of the algorithm was the maximization of $w(Q(\mathbf{x}))$, due to the 'best exploration strategy.' This strategy can of course be replaced by a different exploration strategy. For example, it's possible to simply select random states and actions. This can be done in $\mathcal{O}(1)$ time instead of $\mathcal{O}(n_{blocks})$ time. If this is done, the theoretical order of the algorithm is expected to be $\mathcal{O}(n_{iterations} \log(n_{iterations}))$ or perhaps $\mathcal{O}\left(n_{iterations} \log(n_{iterations})^2\right)$.

To see whether this is accurate, another plot of the run-time can be made, as well as a plot of the run-time divided by $k \log(k)$. (In this case, the natural logarithm has been used. Though note that, if any other logarithm had been used, the result would've been multiplied by a constant, thus not affecting convergence of the plot.) These plots are shown in figure 12-13.

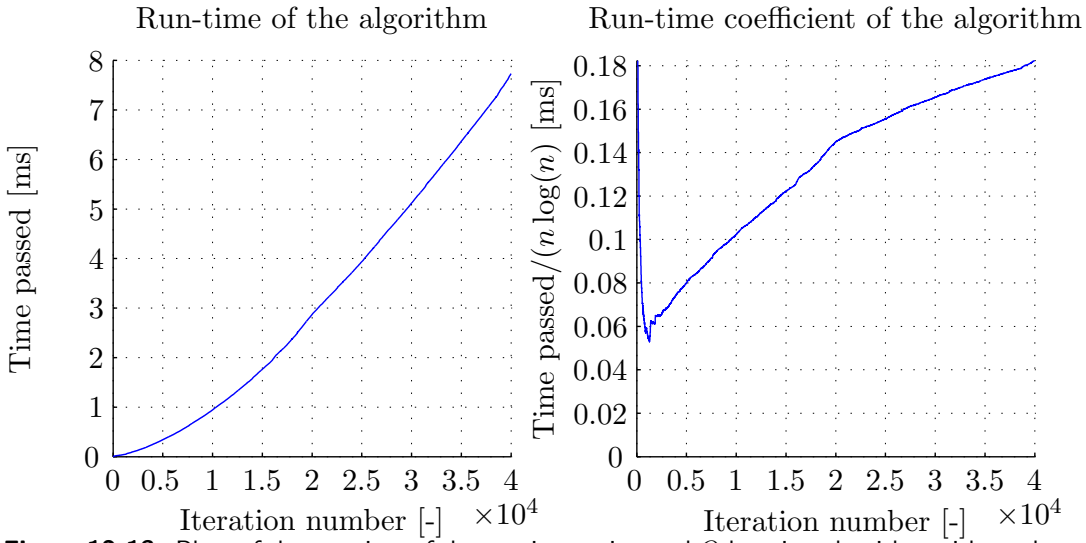

Figure 12-13: Plots of the run-time of the continuous interval Q-learning algorithm with random exploration. On the left is the run-time versus the iteration number. On the right is the run-time divided by $k\ln(k)$.

In the left part of figure 12-13, one can immediately see that the run-time does not quite act quadratically anymore. However, looking at the right part of the figure, it turns out that the order $\mathcal{O}\left(n_{iterations}\log(n_{iterations})\right)$ is not quite correct either. Further research and plots (which are omitted here) have shown that an order of $\mathcal{O}\left(n_{iterations}\log(n_{iterations})^4\right)$ would be appropriate. The reason why this is the case is not certain. Expectations are that the maximization of Q and $\overline{Q}$ is of order $\mathcal{O}\left(\log(n_{iterations})^4\right)$ (or, to be more precise, of order $\mathcal{O}\left(\log(n_{iterations})^{\overline{2D}}\right)$), though good arguments for why this is so have not been found.

In general, a run-time of $\mathcal{O}\left(n\log(n)^4\right)$ is much better than a run-time of $\mathcal{O}\left(n^2\right)$. This gives reason to believe that the 'best exploration strategy' is not so optimal when combined with the function approximator that has been designed in this chapter. Instead, other exploration strategies like random exploration should definitely be kept in mind.

12-5-5 Convergence proof

The algorithm designed in this chapter is a slight modification of the algorithm discussed in the previous chapter. That algorithm already had proven convergence. This adapted algorithm does not have proven convergence yet. This subsection will prove convergence to the optimal value function Q^*.

The main idea of the proof still holds. The volume closed in by the lower and upper bound still always contains Q^*. Furthermore, the remaining volume V is still a quantity that can only decrease. Hence, the algorithm will converge. However, it has not been proven yet that the remaining volume will converge to zero. Maybe it can occur that, at some time, only infinitesimally small updates will be performed, thus not allowing V to converge to zero.

To show that this is cannot be the case, consider an update at time k. A new value for $Q(\mathbf{x_k})$ is then found. Consider the sub-block (i.e. the leaf) that contains $\mathbf{x_k}$. This block has a non-infinitesimal surface area A_{block}. Now two cases can be distinguished. Either the block

containing $\mathbf{x_k}$ can be updated immediately (by a non-infinitesimal amount), or it cannot be. If it can be updated directly, the remaining volume immediately decreases by a non-infinitesimal amount.

If this is not the case – if the block cannot be updated directly – then the block is split up into sub-blocks. According to the algorithm, this continues until there is a sub-block containing $\mathbf{x_k}$ that can be updated (by a non-infinitesimal amount). (Note that this must always occur. After all, if there are sufficient sub-blocks, then the update pyramid can be approximated exactly.) Since this sub-block also has a non-infinitesimal area, the remaining volume again decreases by a non-infinitesimal amount. This implies that the remaining volume always decreases by a non-infinitesimal part, and hence it must converge to zero.

Of course numerical issues may compromise the validity of this proof. If the algorithm is run on a computer with limited numerical accuracy (which in fact holds for all computers), then the remaining volume may converge to a quantity that is not zero (though generally something very close to it). Furthermore, when running the algorithm on a computer, appropriate interval arithmetics must be used, including ideas like outward rounding as discussed in chapter 6, to ensure that no Q-values are missed by the algorithm. If this is done, then the algorithm is guaranteed to converge to something very close to the optimal value function Q^*.

12-6 Application to more complicated problems

So far, the algorithm has only been tested on the simple two-dimensional problem. This section will also see the algorithm applied to a simple three-dimensional problem and of course to the CAP problem discussed earlier in this report. In the end, some general remarks will be given on the application of the algorithm to a system.

12-6-1 A simple three-dimensional problem

To make things a bit more complicated, this subsection considers the **simple three-dimensional problem**. This problem has two state parameters: the position x and the velocity $\dot{x}$. The input equals the acceleration $\ddot{x}$. x falls in the range $[-1, 1]$, $\dot{x}$ falls in the range $[-\frac{1}{2}, \frac{1}{2}]$ and $\ddot{x}$ falls in the range $[-1, 1]$. The time step is $\Delta t = 0.1$ s. Furthermore, $Q_{min} = -4$, $Q_{max} = 1$, $\partial Q/\partial x = 4$, $\partial Q/\partial \dot{x} = 4$, $\partial Q/\partial \ddot{x} = 0.4$ and $\gamma = \frac{1}{2}$. The reward r again equals $|x|$. The goal is thus again to keep the position x near zero. Also, despite its somewhat slower run-time, the best exploration strategy will again be used.

The resulting Q function, after 10^4 iterations, can be seen in figure 12-14. It shows Q as a function of x and $\dot{x}$ for $\ddot{x} = 0$. Note that this time there is no green surface. This is because the optimal value function Q^* could not be easily calculated. Yet still, the red and blue surfaces (the lower and upper bounds) are quite close together, so it's easy to imagine what the optimal value function would be.

It would also be interesting to see whether the average width indeed converges, as the theory says it should. For that, figure 12-15 should be examined. This figure shows that the average width indeed decreases. It is slightly higher than the expected average width, but it still falls well below the bound given by relation (11-2-22).

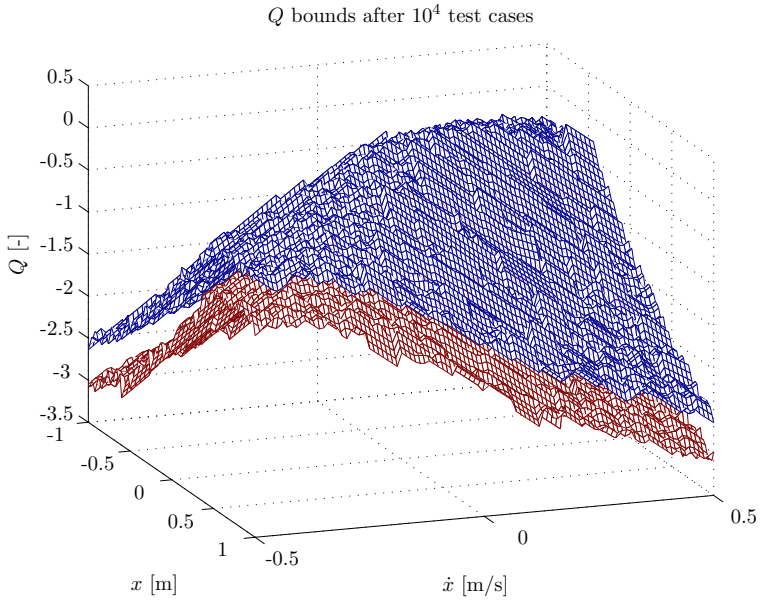

Figure 12-14: Plot of the value function bounds for the simple three-dimensional problem after 10^4 iterations. Q is plotted versus x and $\dot{x}$ with $\ddot{x} = 0$. The red surface indicates $\underline{Q}$ whereas the blue surface indicates $\overline{Q}$.

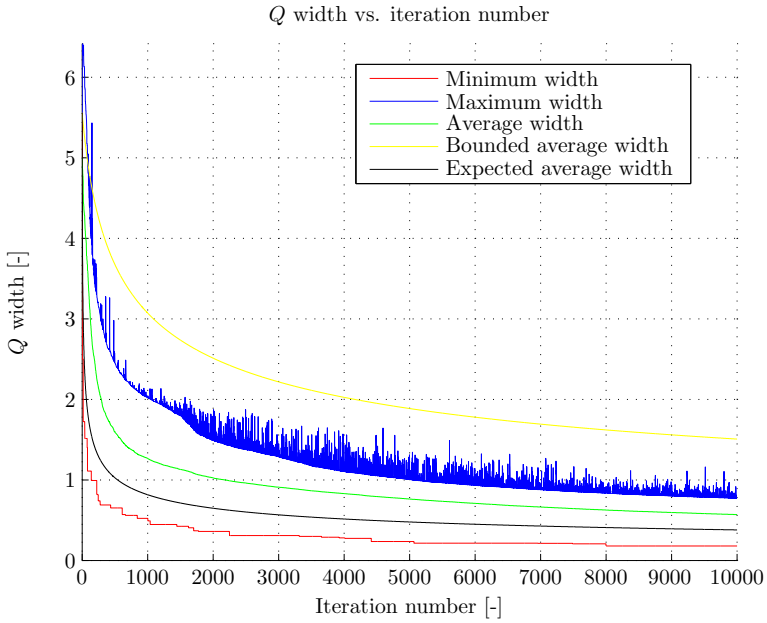

Figure 12-15: Widths of the value function bounds during training of the simple three-dimensional problem. The best exploration strategy has been used.

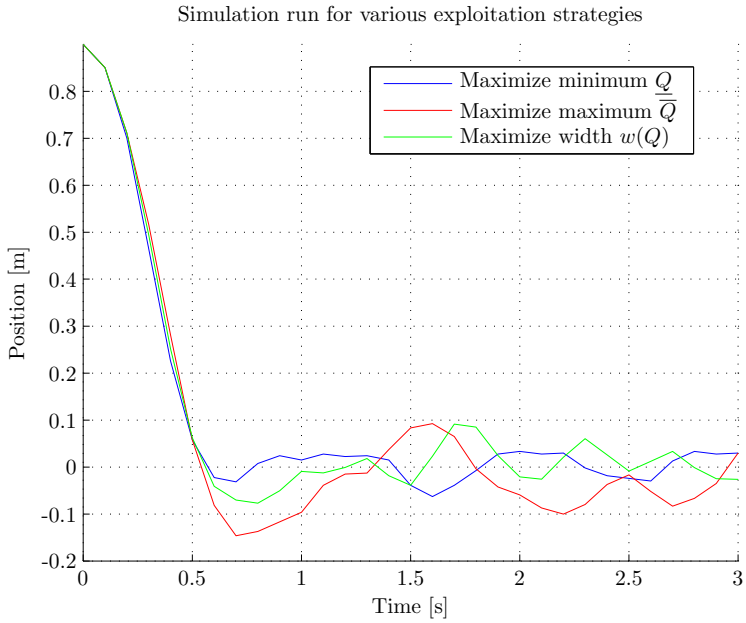

Figure 12-16: Simulation runs of the simple three-dimensional system for various exploitation strategies, after 10^4 iterations of training using the best exploration strategy.

The next question to be answered is: does the RL controller work? That is, can it get the position x to zero and keep it there? To figure that out, first an exploitation strategy has to be chosen. It is possible to select the action with the highest $\overline{Q}$, the action with the highest $\underline{Q}$ or the action with the highest $m(Q)$. (Of course other more complicated exploitation strategies are also possible.) These three strategies have all been applied to a starting state of $x = 0.9$ m and $\dot{x} = 0$ m/s. The resulting system behavior has been plotted in figure 12-16.

Figure 12-16 shows that all strategies are indeed capable of controlling the system. They can definitely move the system towards the zero position. They have not had sufficient training near the zero region to really put the system exactly onto the zero state, but they do keep it near enough. Of course further training in this region would solve this problem.

Finally, it doesn't seem as if any of the three strategies is significantly better than any of the others, so either of the strategies can be used to control the system.

12-6-2 The CAP system

To give the system a real challenge, the CAP system is now considered, as has been defined in section 2-2. This system has four state parameters and one input parameter, thus making it a five-dimensional problem. This problem is considered for $\gamma = 0.5$. A lower value of γ would (as was discussed earlier) not allow the controller to learn adequate long-term behavior, while a higher value would make convergence of w_{av} too slow for any research-like purpose. Furthermore, $Q_{min} = -5$, $Q_{max} = 1$, $dQ/d\alpha = 4$, $dQ/d\dot{\alpha} = 4$, $dQ/dx = 2$, $dQ/d\dot{x} = 2$ and $dQ/dF = 0.2$.

Two test runs have been run, both with 12.000 iterations. One used the best exploration strategy, while the other used random exploration. (That is, every iteration had a fully

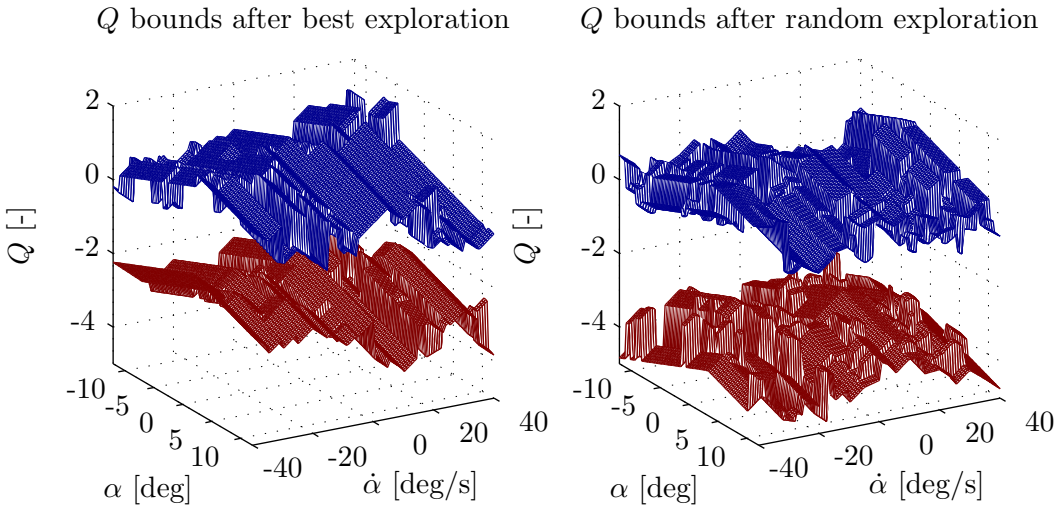

Figure 12-17: The CAP system Q-function after 12.000 iterations. The left Q-function has been trained using the best exploration strategy, while the right Q-function used fully random exploration. Plots are for $x_c = \dot{x}_c = F = 0$.

random state and action.) It must be noted that random exploration was quite a bit faster, as was predicted. The best exploration required 42 minutes, whereas random exploration required 25 minutes to complete the 12.000 iterations.

The resulting Q-function after 12.000 iterations is shown in figure 12-17. Here it can be noted that the Q-function of the best exploration strategy appears much more narrow than the Q-function of random exploration. It thus seems that, despite the longer run-time, the best exploration strategy does work better.

To really compare the two strategies, the width of the Q-function has been plotted as well. This can be seen in figure 12-18. Here a few noticable things can be noticed. First of all, after 12.000 iterations, the best exploration strategy has indeed narrowed more than the random exploration strategy. It has obtained an average width of 2.9, as opposed to 3.2 for random exploration. However, after 8.000 iterations the situation was completely different. Here, the best exploration strategy had $w_{av} = 4.6$ while random exploration had $w_{av} = 3.5$. A significant difference.

This apparently strange phenomenon is caused by the update points selected by the best exploration strategy, combined with the way in which blocks are split up. How it exactly works can be best explained with the help of figure 12-19. Suppose that an update (like the one of figure 12-9) has just been performed. This update is likely to have caused an increase in width $w(Q(\mathbf{x}))$ for some point $\mathbf{x}$. (Especially if D is big, many such points will be present.) Because of this increase in width, the point $\mathbf{x}$ is likely to be the next point that is updated. However, during the initial stages of the algorithm, it is important that blocks are split up, allowing the function approximator to adjust to the updates. When only update points $\mathbf{x}$ from small blocks are selected, then big blocks are not split up. (After all, a block is only split up when it contains the update point $\mathbf{x_k}$ at some time k.) Hence the function approximator cannot adjust properly to updates and the learning is slowed down. That is, until all big blocks have been split up into smaller sub-blocks. Then learning can proceed in its usual effective way. This is the cause of the strange angle in the graph for w_{av}. It occurs

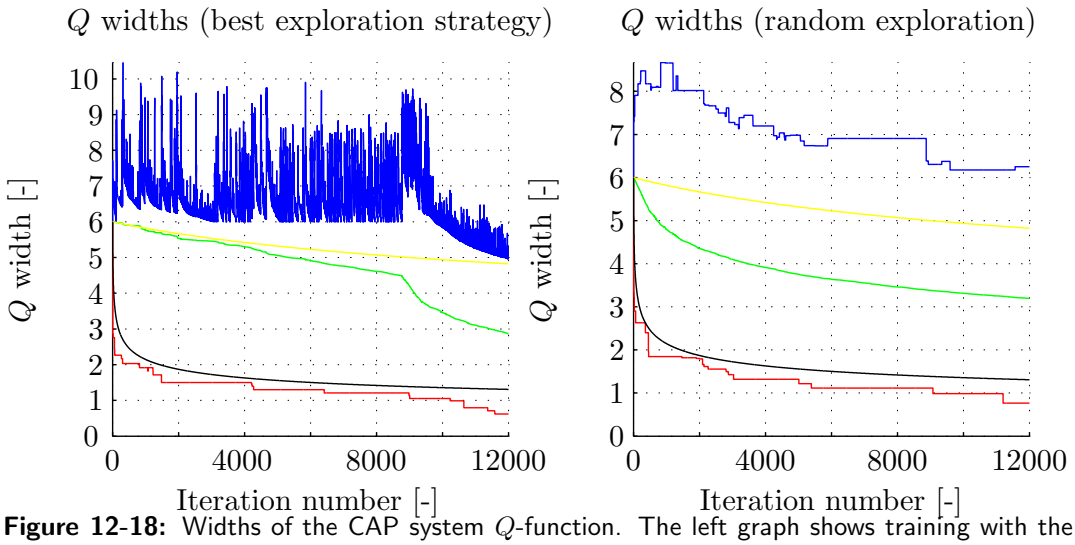

Figure 12-18: Widths of the CAP system Q-function. The left graph shows training with the best exploration strategy, while the right graph displays training with a random strategy. (Line coloring is equivalent to previous graphs in this chapter.)

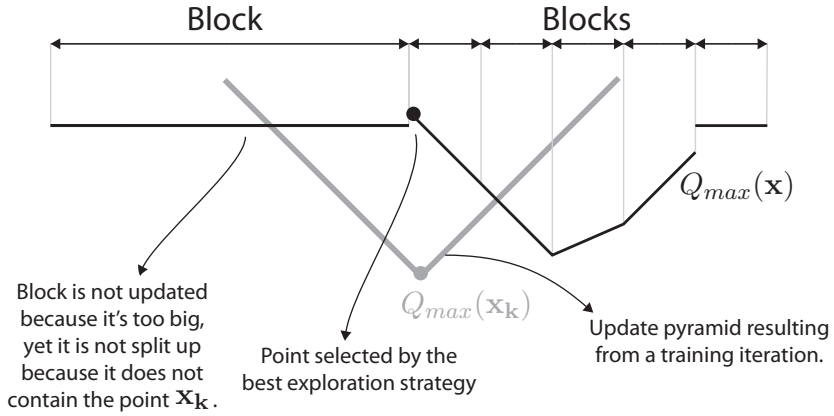

Figure 12-19: When the best exploration strategy is used to select update points, blocks are rarely split up. This significantly slows down learning in the initial stages – another downside of the best exploration strategy.

exactly when all blocks have been split up into sub-blocks. (Note that this strange angle also appeared in figure 12-11, though there it occurred much sooner due to the much smaller state-action-space.)

So how can this problem be prevented? There are plenty of options here. The algorithm to decide when a block is split up can be changed. It is possible to split up all blocks once or twice prior to the running of the algorithm. Or alternatively, it is possible to prevent points $\mathbf{x}$ from having increasing widths $w(Q(\mathbf{x}))$. That is, when an update would cause the width of some point in a block to increase, this block is not updated. This latter option has been implemented, but it proved to be very uneffective. The amount of blocks that were not updated because of it was so high, that the algorithm converged much slower. The other options have not been implemented, so they remain suggestions for further research.

There is one very interesting question remaining: has the resulting Q-function been trained

enough to control the CAP system? Since the average width of the Q-function is still near 3, this seems unlikely. And indeed, this is not the case. When the Q-function is set up to control the CAP system, the pendulum simply falls over. The Q-function has not been trained sufficiently yet to control the system. However, with significantly more training, the Q-function will converge to the optimal value function Q^* and optimal control of the CAP system will be possible.

12-6-3 Controller parameter set-up

During training of the RL controller, several parameters play an important role. It is thus very important to set these parameters accordingly. How this is done depends on the system to which the RL controller is applied.

First of all, the factor γ has a strong influence on the learning performance. Low values of γ make the learning process much faster. However, as has been argued several times earlier in this report, a low value of γ might not always lead to satisfactory performance. Hence, γ needs to be chosen well. Do note that, contrary to certain other types of RL controllers, it is not possible to vary γ during training. This is because changing γ would also change Q^*, which could potentially violate the bounds of the RL controller.

It is also very important to choose the slopes dQ/dx_i well for all parameters x_i. After all, the whole convergence proof is based on the main RL value assumption, and if dQ/dx_i is too low, this assumption is violated. They should thus be chosen big enough. On the other hand, they should also not be chosen too big, as this would significantly reduce the learning performance.

Due to the novelty of this algorithm, no known methods to estimate dQ/dx_i have been developed yet. Choosing values for dQ/dx_i therefore requires good knowledge of the problem at hand, as well as good instincts. Luckily, it is possible to know when too low values for dQ/dx_i have been chosen. If this is the case, then at some time during the algorithm execution, for some point $\mathbf{x}$, the upper bound $\overline{Q}(\mathbf{x})$ will drop below the lower bound $\underline{Q}(\mathbf{x})$. It is easy to check for this during the algorithm execution. This gives the algorithm some integrity. However, if the check is violated, it is first of all not known which slope dQ/dx_i is faulty, and second of all the bounds for Q are known to be invalid, thus requiring the training to start from scratch again. These problems both make it very hard to find accurate values for dQ/dx_i. Good methods to estimate this thus still need to be developed.

The last important parameter is the number of sub-blocks which a block is split up into. When a block is split up, it is split up into n_1 sub-blocks in the direction of parameter x_1, n_2 sub-blocks in the direction of x_2, and so on. The total number of sub-blocks during a split thus is $n_1 n_2 \dots n_D$. It is important to choose the values of n_i well. Choose them too high and the algorithm itself will run slower. Choose them too low and the algorithm will require more iterations because the function approximator cannot adjust to updates well. Furthermore, it is also important to tune n_i to the other numbers n_j. Parameters with big slopes dQ/dx_i need a lot of resolution and thus a high value for n_i. Parameters with small slopes dQ/dx_j need less resolution and can thus do with a smaller value for n_j.

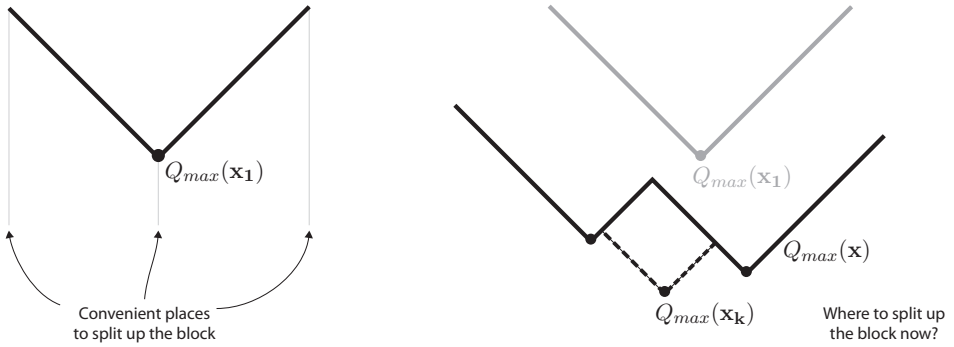

Figure 12-20: A great idea is to vary the points where blocks are split up. However, after several subsequent updates, this poses some problems. Old splitting points have become irrelevant and thus need to be gotten rid of in some way.

12-7 Suggestions for further improvements

The algorithm as it has currently been set up works with proven convergence. However, there are still various improvements that can be made, to either speed up its learning performance or improve its run-time. This section closes off the chapter by giving suggestions for further improvements.

The first suggestion is to experiment with different criteria on when to split up blocks. Currently, blocks are split up when during an update at some point $\mathbf{x_k}$ (A) the block cannot be updated directly, (B) a part of the block could be updated directly and (C) $\mathbf{x_k}$ falls within the block. These criteria are somewhat strict and make it hard for blocks to split up, which has occasionally shown to slow down learning. Perhaps an idea can be thought of that would eliminate the need for criterion (C).

A second suggestion is to vary the way blocks are split up. At the moment, blocks are always split up into a set amount n_i of sub-blocks for each parameter x_i. Maybe a method can be thought of that detects in which direction it is most important to split the block up. The block is then only split up in the direction of this parameter. This would decrease the number of sub-blocks, thus making the algorithm run faster.

Thirdly, one may wonder: 'Why are blocks split up into sub-blocks of fixed length? Isn't it more efficient to split them up at just the right place every time?' Indeed, this may be true. Imagine an update pyramid, as shown in figure 12-20 on the left. If the block is split up exactly at the peak of this update pyramid, then the update pyramid can be approximated exactly by the function approximator. This seems like a great idea.

The problems arrive when more updates in the same region have taken place. In that case old splitting points have become obsolete. If they are not gotten rid of, then the whole function approximator will become inefficient because the splitting points are likely to be clustered in the wrong places. A method thus needs to be developed which can get rid of outdated splitting points. That is, a way needs to be found to merge blocks back together again. If such a thing is possible, then the algorithm can significantly improve. However, merging blocks back together again is more difficult than it seems, and thus requires quite some further

research should it ever be implemented.

Another interesting question is: what happens when the RL agent gets in a state $\mathbf{x}$ which falls outside of the ranges R_i for some parameter x_i? One possible answer to this question would be to argue that such a state is always a terminal state, causing the simulation to end anyway. Another possibility though would be to extend the range R_i that has been violated. That is, more blocks are added in the direction of the parameter x_i. This would enable the RL controller to be very flexible: it isn't bound to a certain range of states and actions, but it can learn for any possible state and action.

A subsequent idea to improve the RL controller concerns the exploration strategy. As was mentioned before, the 'best exploration strategy' apparently isn't always the best strategy, due to its slow run-time for the varying-block algorithm. Different exploration strategies can be tried. Furthermore, the exploration strategies can also take into account certain project constraints. If, for example, during training it is not allowed to get a really low value Q, then a **conservative exploration strategy** can be tried. Only actions whose $\underline{Q}$ value is higher than a given threshold value should be chosen. This would prevent certain potential disasters from occurring, but whether this would also slow down learning, and if so by how much, is a suggestion for further research.

Similar to the exploration strategy, also the exploitation strategy can still be improved. Previous suggestions for exploitation strategies have been to find the action a that maximizes the lower bound $\underline{Q}$, the upper bound $\overline{Q}$, or their mean $m(Q)$. The problem here is that each of these three parameters has a rather jagged shape. There are small peaks and small valleys everywhere. Whether a point $\mathbf{x}$ is a peak or a valley simply depends on whether it was recently updated. This phenomenon reduces the accuracy with which Q^* is approximated. To solve this jaggedness, thus making the approximation of Q^* more accurate, smoothing can be used. For example, Gaussian smoothing can be applied to $m(Q)$, after which this smoothed function is maximized to find the optimal action a. Whether such a thing is possible without serious computational issues and whether it actually does give an improved exploitation strategy are suggestions for further research.

Finally, more attention can also be given to the bounds dQ/dx_i. In this report, it has been assumed that the absolute value of the derivatives dQ/dx_i was bounded by some value. This resulted in pyramid-shaped updates. An alternative is to use bounds of a different format. For example, if $\nabla_{\mathbf{x}}Q$ indicates the gradient vector of Q with respect to $\mathbf{x}$, then it might also be possible to assume the quantity $(\nabla_{\mathbf{x}}Q)^T P(\nabla_{\mathbf{x}}Q)$ is bounded, where P is some conveniently chosen weight matrix. This creates ellipsoid-shaped updates, and although it will be hard to combine this with the current varying-block function approximator, a different function approximator might be developed which can work with ellipsoid-shaped updates.

Chapter 13

Conclusions and recommendations

To close off this thesis report, conclusions are made and recommendations are given.

13-1 Conclusions

At the start of this report, chapter 2 described some issues of continuous reinforcement learning controllers. At the moment, such controllers generally do not have proven convergence. In fact, they occasionally are known to diverge. In short, this project had as goal to solve these issues. A continuous reinforcement learning controller needed to be designed with proven convergence.

To see how far along literature was on this topic, first an extensive literature survey was conducted. Part III summarized the findings. It turned out that in the literature there were only very few instances of continuous reinforcement learning controllers with nonlinear function approximators that had guaranteed convergence. However, none of these controllers were practically applicable, and none of these methods could guarantee convergence to the global optimum either.

During the project, a continuous RL controller has been set up using techniques similar to the ones described in literature. The process of designing this controller has been described in chapter 9. The resulting controller was sometimes able to learn how to control the cart and pendulum system, but sometimes it failed to do so, just like the controllers in literature. Despite several attempts to improve the controller, it was still unclear as to how to ensure convergence of the controller even to a local optimum, let alone the global optimum. Hence, it was decided that it was too complicated to adjust the methods described in literature such that convergence could be guaranteed. Guaranteed convergence to the global optimum of RL controllers with neural network (or similar) function approximators did not seem feasible.

One other goal of this project was to see if reinforcement learning and interval analysis could be successfully combined. Chapter 10 gave a preliminary exploration on this topic. It managed to combine interval analysis and Q-learning into the interval Q-learning method. It

did this by replacing the Q values in the Q-learning method by intervals, and subsequently by narrowing these intervals. This method had proven convergence to the global optimum. However, it still only considered deterministic discrete problems.

After this quite successful initial exploration, chapter 11 continued by expanding this algorithm to the continuous domain. It managed to do this successfully, thanks to the 'main RL value assumption,' which assumed that the derivatives dQ/ds_i and dQ/da_j for every state parameter s_i and action parameter a_j had known bounds. The resulting continuous interval Q-learning algorithm had proven convergence to the global optimum for any deterministic RL problem. And furthermore, also a bound was given on the way which the average width w_{av} of the bounds on Q converged. The bound that was derived equals

$$(w_{av})_k \leq \frac{2w_r}{\left(Dk + \left(\frac{2w_r}{w_0}\right)^D\right)^{\frac{1}{D}}}, \tag{13-1-1}$$

where D is the dimension of the problem (i.e. the number of states and actions combined), w_0 is the initial width of the Q intervals and w_r is a problem-specific constant, depending (among others) on γ, dQ/ds_i, dQ/da_j and the widths of the ranges of the state and action parameters s_i and a_j.

When applied to a practical problem, this algorithm proved to work, although it was very slow. In fact, it was so slow that it was not really practically applicable for any problem with $D \geq 2$. To solve this issue, chapter 12 saw the design of a different function approximator. This function approximator dynamically split up the state-action-space into a number of blocks. For every block, the bounds on Q (denoted by $\underline{Q}$ and $\overline{Q}$) were given a first-order approximation. The resulting algorithm showed to have a run-time of (for some exploration strategies) $\mathcal{O}\left(n_{iterations}^2\right)$ or (for other exploration strategies) $\mathcal{O}\left(n_{iterations} \log(n_{iterations})^{2D}\right)$, which is quite acceptable. This new varying-block continuous interval Q-learning algorithm has a learning performance which is only slightly lower than that of the originally designed continuous interval Q-learning algorithm (that is, it uses its data slightly less efficiently), but its run-time is significantly faster, allowing it to be applied to actual practical problems. Furthermore, this varying-block algorithm also has proven convergence.

In the end, the varying-block continuous interval Q-learning algorithm was applied to several practical problems. For simple two-dimensional and three-dimensional problems (with $D = 2$ and $D = 3$, respectively) the algorithm managed to find a policy close to the optimal policy, even after only short training times. For the cart and pendulum system, which has dimension $D = 5$, this was not the case yet. Half an hour of training (on a reasonably fast home computer) was not enough to ensure a satisfactory policy. However, because the algorithm has guaranteed convergence to the global optimum, it is ensured that with further training the algorithm does converge to the globally optimal policy.

Summarizing, this thesis report has introduced the world's first combination of reinforcement learning and interval analysis. It also introduced the world's first practically feasible continuous RL controller with proven convergence to the global optimum. The key to accomplishing this turned out to be (A) letting go of conventional ways of designing continuous RL controllers, and (B) quantifying the assumption that the value Q of two nearby states s and s' are similar through the main RL value assumption.

13-2 Recommendations

The controller that has been designed in this report has proven convergence to the global optimum. In other words, it works. However, this does not mean that it cannot still be improved. Section 12-7 already provided several suggestions for further improvement. This section will summarize those suggestions, as well as add a few more.

First of all, the varying block continuous interval Q-learning algorithm can be improved by allowing it to use its available data more effectively. To do so, a careful look should be given to the rules that decide both when and how to split up blocks. If blocks are split up in a more effective way, then the efficiency of the algorithm is likely to improve, and especially for higher-dimensional problems this improvement can be quite significant.

The second suggestion is to consider different exploration strategies. A very good exploration strategy has already been suggested, with as most important downside that it is somewhat slow. (It requires the maximization of $w(Q(\mathbf{s}, \mathbf{a}))$, which is computationally a bit difficult.) If a different exploration strategy can be found that doesn't suffer from being very slow, but does have a performance similar to that of the previously mentioned exploration strategy, then the algorithm can run quite a bit faster.

Thirdly, more attention can be given to the bounds of the slopes dQ/ds_i and dQ/da_j. The main assumption of the algorithm is that such bounds are known. How to find these bounds is something that hasn't been discussed a lot in this report. Should the continuous interval Q-learning algorithm be applied to a large number of real-life problems, then a simple and effective way needs to be found in which bounds for these slopes can be obtained. This is another suggestion for further research.

Finally, it must be noted that the continuous interval Q-learning algorithm only works for deterministic systems. For stochastic systems, it is not possible to find accurate bounds that contain the optimal value function Q^* with a 100% certainty. The algorithm therefore doesn't work for such systems. This does not directly mean that stochastic systems cannot benefit from the continuous interval Q-learning algorithm. However, the way in which they might be able to benefit from the algorithm is uncertain. Further research could potentially find ways in which the continuous interval Q-learning algorithm would help to solve stochastic RL problems as well.

Appendix A

Theorems and proofs

Theorem 1

For a positive integer n, a given positive real number x_d and a given positive real number $x_0 \leq x_d$, consider the series x_k (with $k \geq 0$) which decreases according to

$$x_k - x_{k+1} = \left(\frac{x_k}{x_d}\right)^n, \tag{A-1}$$

or equivalently, the ratio between successive terms is given by

$$\frac{x_{k+1}}{x_k} = 1 - \left(\frac{x_k}{x_d}\right)^n. \tag{A-2}$$

This series is upper bounded by

$$x_k^{ub} = \frac{x_d}{\left(nk + \left(\frac{x_d}{x_0}\right)^n\right)^{\frac{1}{n}}}. \tag{A-3}$$

That is, $x_k \leq x_k^{ub}$ for all k from 0 up to ∞.

Proof

First consider the upper bound. Examine the ratio x_{k+1}^{ub}/x_k^{ub}. It equals

$$\frac{x_{k+1}^{ub}}{x_k^{ub}} = 1 - \frac{x_{k+1}^{ub}}{x_d}\left(\frac{x_d}{x_{k+1}^{ub}} - \frac{x_d}{x_k^{ub}}\right) \tag{A-4}$$

$$= 1 - \frac{x_{k+1}^{ub}}{x_d}\left(\left(n(k+1) + \left(\frac{x_d}{x_0}\right)^n\right)^{\frac{1}{n}} - \left(nk + \left(\frac{x_d}{x_0}\right)^n\right)^{\frac{1}{n}}\right). \tag{A-5}$$

Applying theorem 2 (with $x = (x_d/x_0)^n$) results in

$$\frac{x_{k+1}^{ub}}{x_k^{ub}} \geq 1 - \frac{x_{k+1}^{ub}}{x_d} \frac{1}{\left(nk + \left(\frac{x_d}{x_0}\right)^n\right)^{\frac{n-1}{n}}} \tag{A-6}$$

$$= 1 - \frac{x_{k+1}^{ub}}{x_d} \left(\frac{1}{\left(nk + \left(\frac{x_d}{x_0}\right)^n\right)^{\frac{1}{n}}} \right)^{n-1} \tag{A-7}$$

$$= 1 - \frac{x_{k+1}^{ub}}{x_d} \left(\frac{x_k^{ub}}{x_d} \right)^{n-1} \tag{A-8}$$

$$= 1 - \frac{x_{k+1}^{ub} \left(x_k^{ub}\right)^{n-1}}{\left(x_d\right)^n}. \tag{A-9}$$

Working this out further to solve for x_{k+1}^{ub} results in

$$x_{k+1}^{ub} \geq x_k^{ub} - x_{k+1}^{ub} \left(\frac{x_k^{ub}}{x_d} \right)^n \tag{A-10}$$

$$x_{k+1}^{ub} + x_{k+1}^{ub} \left(\frac{x_k^{ub}}{x_d} \right)^n \geq x_k^{ub} \tag{A-11}$$

$$x_{k+1}^{ub} \geq \frac{x_k^{ub}}{1 + \left(\frac{x_k^{ub}}{x_d}\right)^n}. \tag{A-12}$$

(Keep in mind that all quantities are positive. Hence the above operations are valid.) Now consider the case where $x_0 < (n-1)^{-\frac{1}{n}} x_d$. Assume that for some k it holds that $x_k^{ub} \geq x_k$ and $x_k^{ub} < (n-1)^{-\frac{1}{n}} x_d$. (Note that this holds for $k = 0$, as equation (A-3) with $k = 0$ implies $x_0^{ub} = x_0 < (n-1)^{-\frac{1}{n}} x_d$.) Since (according to theorem 5) the above relation is an increasing function in x_k^{ub}, it follows that

$$x_{k+1}^{ub} \geq \frac{x_k}{1 + \left(\frac{x_k}{x_d}\right)^n}. \tag{A-13}$$

(Inserting a lower number into an increasing function will result in a lower function outcome as well.) Also note that

$$x_{k+1} = x_k \left(1 - \left(\frac{x_k}{x_d}\right)^n\right). \tag{A-14}$$

Hence, the ratio x_{k+1}/x_{k+1}^{ub} satisfies

$$\frac{x_{k+1}}{x_{k+1}^{ub}} \leq \frac{x_k \left(1 - \left(\frac{x_k}{x_d}\right)^n\right)}{\frac{x_k}{1+\left(\frac{x_k}{x_d}\right)^n}} \tag{A-15}$$

$$= \left(1 - \left(\frac{x_k}{x_d}\right)^n\right) \left(1 + \left(\frac{x_k}{x_d}\right)^n\right) \tag{A-16}$$

$$= 1 - \left(\frac{x_k}{x_d}\right)^{2n}. \tag{A-17}$$

This is evidently a quantity between 0 and 1, which means that $x_{k+1} \leq x_{k+1}^{ub}$. In other words, $x_k^{ub} \geq x_k$ implies $x_{k+1}^{ub} \geq x_{k+1}$. Since also $x_0^{ub} \geq x_0$, mathematical induction implies that x_k^{ub} is an upper bound for the sequence x_k.

However, there is still a hole in the proof. It is based on the assumption that $x_0 < (n-1)^{-\frac{1}{n}} x_d$. But what happens if $x_0 \geq (n-1)^{-\frac{1}{n}} x_d$? In this case, consider x_1. First it will be shown that $x_1^{ub} > x_1$. Then it will also be shown that $x_1^{ub} < (n-1)^{-\frac{1}{n}} x_d$. Thus, if mathematical induction is applied with x_1^{ub} as starting point, it still works and the proof still holds.

Consider theorem 6 with $x = x_0/x_d$. It implies that

$$1 > \left(\frac{x_0}{x_d}\right)^n \left(1 - \left(\frac{x_0}{x_d}\right)^n\right)^n \left(n + \left(\frac{x_0}{x_d}\right)^n\right), \tag{A-18}$$

$$\frac{x_d^n}{n + \left(\frac{x_0}{x_d}\right)^n} > x_0^n \left(1 - \left(\frac{x_0}{x_d}\right)^n\right)^n, \tag{A-19}$$

$$\left(x_1^{ub}\right)^n > (x_1)^n. \tag{A-20}$$

Hence $x_1^{ub} > x_1$. Next, also consider x_1^{ub} itself. It satisfies

$$x_1^{ub} = \frac{x_d}{\left(n + \left(\frac{x_d}{x_0}\right)^n\right)^{\frac{1}{n}}} \tag{A-21}$$

$$< \frac{x_d}{(n-1)^{\frac{1}{n}}}. \tag{A-22}$$

This indeed proves that $x_1^{ub} < (n-1)^{-\frac{1}{n}} x_d$. This means that x_1^{ub} can be taken as a starting point for induction as well, and hence that the mathematical induction stilll is applicable. The theorem has thus been proven for all provided criteria.

Theorem 2

For integers $n \geq 1$ and $k \geq 0$, as well as any number $x \geq 0$, it holds that

$$\frac{1}{(n(k+1)+x)^{\frac{n-1}{n}}} \leq (n(k+1)+x)^{\frac{1}{n}} - (nk+x)^{\frac{1}{n}} \leq \frac{1}{(nk+x)^{\frac{n-1}{n}}}, \tag{A-23}$$

where equality only occurs if $n = 1$.

Proof

The above relation consists of two inequalities. First the right-hand part will be proven, which is followed by a proof of the left-hand part.

Consider the term

$$(nk+1+x)^n = ((nk+x)+1)^n. \tag{A-24}$$

Using a binomial expansion, this can be worked out as

$$(nk+1+x)^n = \binom{n}{n}(nk+x)^n + \binom{n}{n-1}(nk+x)^{n-1} + \ldots + \binom{n}{0}(nk+x)^0 . \quad \text{(A-25)}$$

All the terms in this expansion are positive. Removing terms would thus decrease the above quanity. Hence it follows that

$$(nk+1+x)^n \geq \binom{n}{n}(nk+x)^n + \binom{n}{n-1}(nk+x)^{n-1}, \quad \text{(A-26)}$$

where equality only occurs if $n = 1$ (and thus no terms are removed). The above relation can be worked out further, such that

$$\begin{aligned}
(nk+1+x)^n &\geq \binom{n}{n}(nk+x)^n + \binom{n}{n-1}(nk+x)^{n-1} && \text{(A-27)}\\
&= (nk+x)^n + n(nk+x)^{n-1} && \text{(A-28)}\\
&= (nk+x)(nk+x)^{n-1} + n(nk+x)^{n-1} && \text{(A-29)}\\
&= (nk+x+n)(nk+x)^{n-1} && \text{(A-30)}\\
&= (n(k+1)+x)(nk+x)^{n-1} . && \text{(A-31)}
\end{aligned}$$

Taking the n^{th} root of this relation gives

$$nk+1+x \geq (n(k+1)+x)^{\frac{1}{n}}(nk+x)^{\frac{n-1}{n}} . \quad \text{(A-32)}$$

Working things out further results in

$$\begin{aligned}
1 &\geq (n(k+1)+x)^{\frac{1}{n}}(nk+x)^{\frac{n-1}{n}} - (nk+x), && \text{(A-33)}\\
\frac{1}{(nk+x)^{\frac{n-1}{n}}} &\geq (n(k+1)+x)^{\frac{1}{n}} - (nk+x)^{\frac{1}{n}} . && \text{(A-34)}
\end{aligned}$$

This proves the right-hand part of the theorem. (Note that equality still only holds when $n = 1$.)

The proof of the left-hand part is similar, although one extra complication arises. Consider the term

$$(n(k+1)-1+x)^n = ((n(k+1)+x)-1)^n. \quad \text{(A-35)}$$

This can be worked out as the alternating sum

$$(n(k+1)-1+x)^n = \binom{n}{n}(n(k+1)+x)^n - \binom{n}{n-1}(n(k+1)+x)^{n-1} + \ldots \pm \binom{n}{0}(nk+x)^0 . \quad \text{(A-36)}$$

Consider all but the first two terms of this series. According to theorem 3 (with $a = n(k+1)+x$), the sum of all these terms is non-negative. This implies that

$$(n(k+1)-1+x)^n \geq \binom{n}{n}(n(k+1)+x)^n - \binom{n}{n-1}(n(k+1)+x)^{n-1}, \quad \text{(A-37)}$$

where equality only occurs if $n = 1$. The relation can then again be worked out further, according to

$$\begin{aligned}
(n(k+1)-1+x)^n &\geq \binom{n}{n}(n(k+1)+x)^n - \binom{n}{n-1}(n(k+1)+x)^{n-1} && \text{(A-38)}\\
&= (n(k+1)+x)^n - n(n(k+1)+x)^{n-1} && \text{(A-39)}\\
&= (n(k+1)+x)(n(k+1)+x)^{n-1} - n(n(k+1)+x)^{n-1} && \text{(A-40)}\\
&= (n(k+1)+x-n)(n(k+1)+x)^{n-1} && \text{(A-41)}\\
&= (nk+x)(n(k+1)+x)^{n-1}. && \text{(A-42)}
\end{aligned}$$

Taking the n^{th} root of this relation gives

$$n(k+1)-1+x \geq (nk+x)^{\frac{1}{n}}(n(k+1)+x)^{\frac{n-1}{n}}. \tag{A-43}$$

Working things out further then results in

$$\begin{aligned}
1 &\leq (n(k+1)+x) - (nk+x)^{\frac{1}{n}}(n(k+1)+x)^{\frac{n-1}{n}}, && \text{(A-44)}\\
\frac{1}{(n(k+1)+x)^{\frac{n-1}{n}}} &\leq (n(k+1)+x)^{\frac{1}{n}} - (nk+x)^{\frac{1}{n}}. && \text{(A-45)}
\end{aligned}$$

This proves the left-hand part of the theorem, and thus the proof of the theorem has been completed.

Theorem 3

Consider the binomial expansion of $(a-1)^n$ for a certain number $a \geq 1$ and integer $n \geq 1$,

$$(a-1)^n = \binom{n}{n}a^n - \binom{n}{n-1}a^{n-1} + \binom{n}{n-2}a^{n-2} - \binom{n}{n-3}a^{n-3} + \ldots \pm \binom{n}{0}a^0. \tag{A-46}$$

The sum of all but the first two elements in this expansion is greater than or equal to zero. That is,

$$\binom{n}{n-2}a^{n-2} - \binom{n}{n-3}a^{n-3} + \ldots \pm 1 \geq 0, \tag{A-47}$$

or equivalently,

$$(a-1)^n + na^{n-1} \geq a^n. \tag{A-48}$$

Furthermore, equality only occurs if $n = 1$.

Proof

First consider the case where $n \geq a$. In this case,

$$(a-1)^n + na^{n-1} \geq (a-1)^n + a^n \geq a^n. \tag{A-49}$$

Equality only occurs if (for the left hand side) $n = a$ and (for the right hand side) $a = 1$. Since it was assumed that $n \geq a$, equality thus only occurs whenever $n = 1$.

Second, consider the case where $n < a$. Now examine the series

$$\binom{n}{n-2}a^{n-2} - \binom{n}{n-3}a^{n-3} + \ldots \pm 1. \tag{A-50}$$

Or to be more precise, consider two subsequent terms in this series, of which the first is positive and the second is negative. In other words, for some even integer k (subject to $2 \leq k \leq n-1$), examine

$$\binom{n}{n-k}a^{n-k} - \binom{n}{n-k-1}a^{n-k-1}. \tag{A-51}$$

Using theorem 4 it can be shown that

$$\begin{aligned}
\binom{n}{n-k}a^{n-k} - \binom{n}{n-k-1}a^{n-k-1} &= \binom{n}{n-k}\left(a^{n-k} - \frac{\binom{n}{n-k-1}}{\binom{n}{n-k}}a^{n-k-1}\right) && \text{(A-52)}\\
&\geq a^k - na^{k-1} && \text{(A-53)}\\
&> a^k - a^k = 0. && \text{(A-54)}
\end{aligned}$$

Hence every pair of numbers in the series is bigger than zero. This implies that the sum of all these pairs is also bigger than zero. (Note that, when n is odd, the last number of the series is not considered, but since this number always equals 1, it is also bigger than zero.) Concluding, the series is always bigger than zero. The only exception occurs when the series consists of no terms whatsoever, which in turn is the case when $n = 1$. In this case, and only in this case, does the series equal zero. This proves the theorem.

Theorem 4

For every positive integer n and positive integer k with $1 \leq k \leq n$, it holds that

$$\frac{\binom{n}{k-1}}{\binom{n}{k}} \leq n, \tag{A-55}$$

where equality occurs if and only if $k = n$.

Proof

Consider the left-hand side of the above relation. It can be worked out using the definition

$$\binom{a}{b} = \frac{a!}{b!(a-b)!}. \tag{A-56}$$

This turns the left-hand side into

$$\frac{\frac{n!}{(k-1)!(n-k+1)!}}{\frac{n!}{k!(n-k)!}} = \frac{k!(n-k)!}{(k-1)!(n-k+1)!} = \frac{k!}{(k-1)!}\frac{(n-k)!}{(n-k+1)!} = \frac{k}{n-k+1}. \tag{A-57}$$

Now note that, because of $k \leq n$,

$$\frac{n+1}{k} - 1 \geq \frac{n+1}{n} - 1 = \frac{1}{n}, \tag{A-58}$$

where equality only occurs when $k = n$. Hence,

$$\frac{k}{n-k+1} = \frac{1}{\frac{n+1}{k} - 1} \leq n, \tag{A-59}$$

where still equality only occurs if $k = n$. This proves the theorem to be proven.

Theorem 5

Consider the function

$$y(x) = \frac{x}{1 + \left(\frac{x}{x_d}\right)^n} \tag{A-60}$$

for some integer $n \geq 1$ and some real numbers $x \geq 0$ and $x_d \geq 0$. If $n = 1$, this always is an increasing function. If $n \geq 2$ then the function is increasing for

$$x < (n-1)^{-\frac{1}{n}} x_d. \tag{A-61}$$

Proof

The derivative of y is given by

$$\frac{dy}{dx} = \frac{\left(1 + \left(\frac{x}{x_d}\right)^n\right) - n\left(\frac{x}{x_d}\right)^n}{\left(1 + \left(\frac{x}{x_d}\right)^n\right)^2}. \tag{A-62}$$

Because the denominator is positive, this quantity is positive whenever the numerator is positive. Note that for $n = 1$ this always holds. (The numerator then equals 1.) For $n \geq 2$, the numerator is positive whenever

$$1 + \left(\frac{x}{x_d}\right)^n - n\left(\frac{x}{x_d}\right)^n > 0, \tag{A-63}$$

$$1 - (n-1)\left(\frac{x}{x_d}\right)^n > 0, \tag{A-64}$$

$$\frac{1}{n-1} > \left(\frac{x}{x_d}\right)^n, \tag{A-65}$$

$$\left(\frac{1}{n-1}\right)^{\frac{1}{n}} > \frac{x}{x_d}. \tag{A-66}$$

$$\tag{A-67}$$

This directly implies the theorem to be proven.

Theorem 6

For any real number $x \in [0, 1]$ and any integer $n \geq 1$ it holds that

$$x^n(1-x^n)^n(n-1) \leq x^n(1-x^n)^n(n+x^n) < 1. \tag{A-68}$$

Proof

The left part of the inequality is trivial, so only the right part will be proven. First consider the quantity $y = x(1-x^n)$. Its derivative with respect to x equals

$$\frac{dy}{dx} = (1-x^n) - nx^n. \tag{A-69}$$

This equals zero whenever

$$x = \left(\frac{1}{n+1}\right)^{\frac{1}{n}}. \tag{A-70}$$

It can be verified (using a second derivative) that this point is indeed a maximum of y. The corresponding maximum value of y equals

$$y_{max} = \left(\frac{1}{n+1}\right)^{\frac{1}{n}}\left(1-\frac{1}{n+1}\right) = \left(\frac{1}{n+1}\right)^{\frac{1}{n}}\left(\frac{n}{n+1}\right). \tag{A-71}$$

It thus follows that for every $x \in [0, 1]$,

$$x^n(1-x^n)^n \leq y_{max}^n = \frac{1}{n+1}\left(\frac{n}{n+1}\right)^n. \tag{A-72}$$

Now consider the quantity $n + x^n$. Since $x \leq 1$, it holds that $n + x^n \leq n + 1$. Therefore,

$$x^n(1-x^n)^n(n+x^n) \leq \frac{1}{n+1}\left(\frac{n}{n+1}\right)^n(n+1) = \left(\frac{n}{n+1}\right)^n < 1. \tag{A-73}$$

This proves the theorem.

Appendix B

Thoughts on artificial intelligence

During the course of this project, a lot of thoughts have been spent on making systems more intelligent, and more like humans. Many artificial intelligence (AI) techniques, like reinforcement learning and neural networks, are inspired by the way that humans think. However, it turned out that for this project this was not beneficial. After all, the goal of this project was to make a self-learning controller that always converges to the global optimum, yet it is known that humans do not possess this property. In fact, when humans learn a complicated skill on their own, without a proper teacher, they often learn sub-optimal techniques. The continuous interval Q-learning algorithm designed during this project is totally different from the way that humans think, yet it does have convergence to the global optimum.

Still, many thoughts have been spent on how humans learn and think. To not let these thoughts go to waste, they are summarized in this appendix. This appendix aims to answer the question what properties a controller should have before it can become even remotely as smart as a human being. Several properties will be described and their links will be elaborated on.

The first property of a human-like system is **conceptualization**: the ability to form concepts. This is the property which has inspired the AI field of pattern recognition. When a human being sees many objects that all have four legs, like to chase balls or frisbees and regularly bark, he puts them all into one big category called a concept. Then, when another creature appears that also has four legs and barks, it is recognized to fall in this specific concept. Similarly, if an aircraft controller regularly encounters low-frequency oscillations of the pitch angle, combined with variations in the aircraft velocity, it groups these oscillations together into a concept as well. This process of recognizing that some phenomenon is part of a group of other related phenomena is fundamental to the way the human mind works.

But humans don't just conceptualize. They also do **concept branching**. When a human being spends a lot of time with a certain concept, this concept branches out into sub-concepts. For example, when a human being spends a lot of time among dogs, he might start to notice subtle differences between dogs. Some dogs are small, have white puffy hair and a tiny

tail. Other dogs are big, have a brown/black color and pointy ears. Through noticing these differences, the human mind makes a sub-division within the concept dogs. There are dogs that are poodles and there are dogs that are German shepherds. These sub-concepts have all the properties that their parent concepts have, but they are more detailed and can thus only be formed when sufficient experience with the original concept is present. Furthermore, these concepts (and sub-concepts) are also dynamic. Initially, you might believe that all birds fly, but when you then suddenly discover penguins or ostriches, you have to readjust your concept of birds. In this way, concepts are adjusted on a daily basis.

So what does the human mind do with these concepts? Most importantly, it forms **relationships** between these concepts. This is the property which has inspired the logic-based computer systems that gave birth to the field of 'artificial intelligence.' Imagine that you put your foot down on the floor and simultaneously you hear a loud knock. Your mind then immediately establishes the relationship between the concepts 'putting your foot down' and 'hearing a knock.' If you put your foot down again, and you hear a knock again, this relation is strengthened further. But if you then put your foot down and don't hear a knock, the relationship is weakened. In fact, when you then find out that your neighbour was knocking on your door, coincidentally just at the times when you put down a foot, the relationship is blown away.

This whole process of forming relationships is something which the mind is really active at. In fact, it sometimes does this a bit too much. If you yell at someone, and a few seconds later a picture frame falls from the wall, then your mind creates a link between these two events, even though such a link in reality does not exist. These faulty relationships are often known as 'superstitions.' It is important to keep in mind here that, should a human-like controller ever be designed, then it should be accepted that such human-like disadvantages are also present in that controller.

It should be noted here that relationships are often (if not always) stochastic. The concepts 'yelling at someone' and 'making that person angry' are of course connected, but there's not a 100% relation between these two concepts. Some people simply don't get angry when people yell at them. Furthermore, relationships often have quantifications attached to them as well. The concept of 'throwing a ball' is connected to 'the ball lands somewhere in front of you.' However, throwing a ball faster implies that the ball lands further. The exact relationship between 'ball velocity' and 'landing distance' is therefore a quantified relationship.

A fourth very important property of the human mind is **concept valuation**. This is the property which has inspired reinforcement learning. Human beings always have certain things they like. One thing which (almost) all humans aspire to, is to feel good and healthy. This concept of 'feeling well' is something they value highly. Many young people also have the link that 'eating sugary food' helps them to 'feel good.' (After all, these sugars give young people lots of energy.) There is thus a relationship between these two contepts. Because of this, a lot of young people like to eat sugary food.

But there is more. Initially, young people like to eat sugary food because of the link that it makes them feel good. As time passes, the concept of 'eating sugary food' gains value of its own. It's something that these people value by itself, even without any links. And as it turns out, later on in life eating sugary food isn't so good anymore. For older people, the link between 'eating sugary food' and 'feeling healty' not only disappears, it is also reversed. (For people past their teenager years, eating sugar often does not result in more energy but

in less, because it all needs to be digested.) Yet still human beings well into their twenties or thirties often like to eat sugary food. It's only after many years that the reversion of the link manages to reduce the concept value of 'eating sugary foods' enough that people stop liking it so much.

Summarizing, the human mind is able to form concepts, split up concepts into sub-concepts, recognize phenomena as being part of a concept, form relationships between concepts, and value concepts, all in a very dynamic way. Because of this, it is able to quickly learn how to cope with unexpected circumstances. When some unexpected event happens, the mind quickly makes a new concept out of the event, it establishes links between this new concept and older well-known concepts and it then uses the values of these older concepts to guess how positive this new unexpected event is.

The claim that is being made in this appendix is that, should a computer program ever be designed that works in a way similar to the human mind, then it must possess all of the properties/ideas that have been mentioned in this appendix. This does not mean that these ideas are the key to intelligence – if this report has shown only one thing, then it's that it's also possible to create intelligent systems that work totally different from the human mind – but it does mean that any attempt at mimicking human intelligence should incorporate as many of these ideas as possible. After all, they are part of the key to how the human mind works.

Bibliography

Agostini, A. G., & Celaya Llover, E. (2010). Reinforcement learning for robot control using probability density estimations. *7th International Conference on Informatics in Control, Automation and Robotics*, 160-168.

Aviation Safety Network. (2009). *ASN Aircraft accident de Havilland Canada DHC-8-402 Q400 N200WQ Buffalo Niagara International Airport, NY (BUF).* Available from `http://aviation-safety.net/database/record.php?id=20090212-0`

Babuška, R. (2010). *Knowledge-based control systems.* Delft University of Technology.

BEA (Bureau d'Enquêtes et d'Analyses pour la sécuritéde l'aviation civile). (2011, July). *Third interim report on flight af 447.* Available from `http://www.bea.aero/docspa/2009/f-cp090601e3.en/pdf/f-cp090601e3.en.pdf`

Beitelspacher, J., Fager, J., Henriques, G., & Mcgovern, A. (2006). *Policy gradient vs. value function approximation: A reinforcement learning shootout.*

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-dynamic programming.* Athena Scientific.

Bhatnagar, S., Sutton, R. S., Ghavamzadeh, M., & Lee, M. (2009). Natural actor critic algorithms. *Automatica*, *45*(11), 2471-2482.

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems 7* (p. 369-376).

Bradtke, S. J. (1993). Reinforcement learning applied to linear quadratic regulation. In *Advances in neural information processing systems 5, [nips conference]* (p. 295-302).

Daw, N. D., & Frank, M. J. (2009). Reinforcement learning and higher level cognition. *Cognition*, *113*(3), 259-261.

de Weerdt, E., Chu, Q. P., & Mulder, J. A. (2009). Neural network output optimization using interval analysis. *IEEE Transactions on Neural Networks*, *20*(4), 638-653.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Mach. Learn.*, *43*(1-2), 7–52.

Frank, J., Mannor, S., & Precup, D. (2008). Reinforcement learning in the presence of rare events. In *Proceedings of the 25th international conference on machine learning* (p. 336-343).

Gordon, G. J. (2001). Reinforcement learning with function approximation converges to a region. In *Advances in neural information processing systems* (p. 1040-1046).

Graupe, D. (1997). *Principles of artificial neural networks.* World Scientific Publishing Co. Pte. Ltd.

Hawkins, J., & Blakeslee, S. (2004). *On intelligence.* Henry Holt and Company, LLC.

Haykin, S. S. (1999). *Neural networks.* Prentice Hall.

Huys, Q. J. M., & Dayan, P. (2009). A bayesian formulation of behavioral control. *Cogni*, *113*(3), 314-328.

Irodova, M., & Sloan, R. H. (2005). Reinforcement learning and function approximation. *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2005 - Recent Advances in Artifical Intelligence*, 455-460.

Lagoudakis, M. G., Parr, R., & Bartlett, L. (2003). Least-squares policy iteration. *Journal of Machine Learning Research*, *4*, 1107-1149.

Levin, E., Tishby, N., & Solla, S. A. (1989). A statistical approach to learning and generalization in layered neural networks. In *Proceedings of the second annual workshop on computational learning theory* (pp. 245–260).

Melo, F. S., Meyn, S. P., & Ribeiro, M. I. (2008). An analysis of reinforcement learning with function approximation. In *Proceedings of the 25th international conference on machine learning* (p. 664-671).

Merke, A., & Schoknecht, R. (2004). Convergence of synchronous reinforcement learning with linear function approximation. In *Proceedings of the twenty-first international conference on machine learning* (p. 75-80).

Moore, R. E., Baker Kearfott, R., & Cloud, M. J. (2009). *Introduction to interval analysis* (Third ed.). SIAM - Society for Industrial and Applied Mathematics, Philadelphia.

Motta Salles Barreto, A. da, & Anderson, C. W. (2008). Restricted gradient-descent algorithm for value-function approximation in reinforcement learning. *Artificial Intelligence*, *172*(4-5), 454-482.

Pink, D. H. (2010). *Drive.* Canongate Books.

Prokhorov, D., & Wunsch, D. (1997). Adaptive critic designs. *IEEE Transactions on Neural Networks*, *8*, 997-1007.

Qiang, W., & Zhongly, Z. (2011). Reinforcement learning model, algorithms and its application. *International Conference on Mechatronic Science, Electric Engineering and Computer, Jilin, China*, *1*, 1143-1146.

Raad voor de Luchtvaart (Netherlands Aviation Safety Board). (1994). *Aircraft accident report 92-11 - el al flight 1862* (Tech. Rep.). Ministerie van Verkeer en Waterstaat.

Russell, S. J., & Norvig, P. (2003). *Artificial intelligence* (Second ed.). Prentice Hall.

Schiller, U. D. (2003). *Analysis and comparison of algorithms for training recurrent neural networks.* Unpublished master's thesis, The Neuroinformatics Group - Faculty of Technology - University of Bielefeld.

Stone, P., & Veloso, M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robotics*, *8*(3), 345-383.

Sutton, R. S., & Barto, A. H. (1998). *Reinforcement learning.* MIT Press.

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). *Policy gradient methods for reinforcement learning with function approximation* (Tech. Rep.). AT&T Labs.

Swaab, D. (2010). *Wij zijn ons brein.* Uitgeverij Atlas Contact.

Thrun, S., & Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. In *Proceedings of the fourth connectionist models summer school.*

Tsitsiklis, J. N., & van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*(5), 674-690.

van Kampen, E. (2006). *Continuous adaptive critic flight control using approximated plant dynamics*. Unpublished master's thesis, Delft University of Technology.

van Kampen, E. (2010). *Global optimization using interval analysis*. Unpublished doctoral dissertation, Delft University of Technology.

Webb, A. R. (2002). *Statistical pattern recognition* (Second ed.). Wiley & Sons.

Whiteson, S., & Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *J. Mach. Learn. Res.*, *7*, 877-917.

Zalzala, A., & Morris, A. (1996). *Neural networks for robotic control*. Ellis Horwood.

Zheng, Y., Luo, S., & Lv, Z. (2006). Control double inverted pendulum by reinforcement learning with double cmac network. In *Proceedings of the 18th international conference on pattern recognition - volume 04* (p. 639-642).

Index