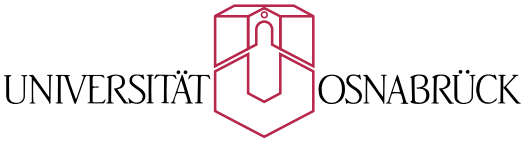


Institut für Informatik
Neuroinformatics Group

Reinforcement Learning with Recurrent Neural Networks

Dissertation
zur
Erlangung der Doktorwürde
der Universität Osnabrück
– Fachbereich Mathematik/Informatik –

Vorgelegt von
Herrn Anton Maximilian Schäfer

Osnabrück, den 31.10.2008

Supervisors: Prof. Dr. Martin Riedmiller, University of Osnabrück
Dr. Hans-Georg Zimmermann, Siemens AG

Abstract

Controlling a high-dimensional dynamical system with continuous state and action spaces in a partially unknown environment like a gas turbine is a challenging problem. So far often hard coded rules based on experts' knowledge and experience are used. Machine learning techniques, which comprise the field of reinforcement learning, are generally only applied to sub-problems. A reason for this is that most standard reinforcement learning approaches still fail to produce satisfactory results in those complex environments. Besides, they are rarely data-efficient, a fact which is crucial for most real-world applications, where the available amount of data is limited.

In this thesis recurrent neural reinforcement learning approaches to identify and control dynamical systems in discrete time are presented. They form a novel connection between recurrent neural networks (RNN) and reinforcement learning (RL) techniques. Thereby, instead of focusing on algorithms, neural network architectures are put in the foreground.

RNN are used as they allow for the identification of dynamical systems in form of high-dimensional, non-linear state space models. Also, they have shown to be very data-efficient. In addition, a proof is given for their universal approximation capability of open dynamical systems. Moreover, it is pointed out that they are, in contrast to an often cited statement, well able to capture long-term dependencies.

As a first step towards reinforcement learning, it is shown that RNN can well map and reconstruct (partially observable) Markov decision processes. In doing so, the resulting inner state of the network can be used as a basis for standard RL algorithms. This so-called hybrid RNN approach is rather simple but showed good results for a couple of applications. The further developed recurrent control neural network combines system identification and determination of an optimal policy in one network. It does not only learn from data but also integrates prior knowledge into the modelling in form of architectural concepts. Furthermore, in contrast to most RL methods, it determines the optimal policy directly without making use of a value function. This distinguishes the approach also from other works on reinforcement learning with recurrent networks.

The methods are tested on several standard benchmark problems. In addition, they are applied to different kinds of gas turbine simulations of industrial scale.

Acknowledgement

This work was supported by Siemens AG, Corporate Technology, department Learning Systems in Munich.

I especially thank Prof. Dr. Martin Riedmiller for his supervision, help and support. I further thank Dr. Thomas Runkler, head of the department Learning Systems, Siemens AG, for giving me the opportunity to develop and to write my thesis within this scientifically highly competent research group.

Besides my thanks goes to all my colleagues for their support and patience during the preparation of my thesis. The multiple discussions and meetings formed a great enrichment to my research. In particular I want to thank my two advisors at Siemens AG, Dr. Hans-Georg Zimmermann and Dr. Steffen Udluft, for their continuous encouragement and the advice they gave me.

The computations in this thesis were performed on the neural network modelling software SENN (*Simulation Environment for Neural Networks*), which is a product of Siemens AG. Thanks to Dr. Christoph Tietz for its continuous adaption and extension to the new requirements of the developed networks.

Contents

List of Figures

List of Tables

List of Abbreviations

AHC	adaptive heuristic critic
CT	combustion tuning
DP	dynamic programming
ET	emission tuning
EET	extended emission tuning
FFNN	feedforward neural network
IGV	inlet guide vane
LSTM	long short-term memory
MDP	Markov decision process
POMDP	partially observable Markov decision process
PS	prioritised sweeping
RL	reinforcement learning
RMS	pressure intensities in the combustion chamber
RNN	recurrent neural network
RPS	RNN based prioritised sweeping
RQ	RNN based Q-learning
RCNN	recurrent control neural network
STD	standard deviation
TD	temporal difference

”To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science.”

Albert Einstein, 1879 – 1955

CHAPTER 1

Introduction

Reinforcement learning and control problems of industrial scale, like the control of a gas turbine, are generally high-dimensional, extensive and only partially observable. Due to the large amount of different inter-depending parameters and information, which is still insufficient to fully describe the problem, human beings are hardly able to find an optimal solution.

Therefore the field of machine learning, in particular approximate dynamic programming or reinforcement learning, aims at developing (self-adapting) algorithms, which are able to learn how to control a system out of the available data. Many different solution methods have been proposed during the last 50 years. The prevalent ones are well summarised in the books of Bertsekas [10, 11], and Sutton and Barto [89]. Still, besides other restrictions due to Bellman’s ”curse of dimensionality” [7], most of them fail to produce good results for high-dimensional and partially observable problems with continuous state and action spaces where further the available amount of data is limited. In order to cover these issues, sophisticated and data-efficient learning methods are required, which are able to deal with high-dimensionality and non-linearities, and take short and long-term influences into account.

1.1 Reinforcement Learning Problems

Reinforcement learning (RL) (chap. 2) is an ideal approach to solve optimal control problems by learning a policy, which maximises a desired outcome. It basically considers a controller or agent and the environment, with which the controller interacts by carrying out different actions. For each interaction the controller can observe the outcome of its action. In other words, the agent gets a positive or negative reward, which is used to optimise its action selection or control policy, i.e., its future actions based on the respective state.

Throughout this thesis a reinforcement learning or control problem is regarded as an open, time-discrete dynamical system with a correspondent additive reward function [10]. Those systems can be used to describe most technical or economical real-world applications, which are by construction mainly deterministic. Hereby, it is assumed that stochasticity basically occurs due to partial observability.

Let therefore be $S \subseteq \mathbb{R}^J$ the system's (real) environmental state space with states $\mathbf{s}_t \in S$, $X \subseteq \mathbb{R}^I$ the space of observables $\mathbf{x}_t \in X$, which generally is a subspace of S, and $U \subseteq \mathbb{R}^K$ the control or action space with control or action parameters $\mathbf{u}_t \in U$ (with $I, J, K \in \mathbb{N}$ and $t = 1, \ldots, \infty$). The dynamical system can then be described by the following set of equations

$$\begin{array}{rcl} \mathbf{s}_{t+1} & = & g(\mathbf{s}_t, \mathbf{u}_t) \\ \mathbf{x}_t & = & h(\mathbf{s}_t) \end{array} \tag{1.1}$$

with $g : \mathbb{R}^J \times \mathbb{R}^K \to \mathbb{R}^J$ and $h : \mathbb{R}^J \to \mathbb{R}^I$ being two arbitrary (non-linear) functions. This corresponds to the assumption that the next internal state $\mathbf{s}_{t+1}$ evolves from the current one $\mathbf{s}_t$, influenced by the current action $\mathbf{u}_t$. Further, the observables $\mathbf{x}_t$ develop out of the current internal state $\mathbf{s}_t$.[1]

Nearly all reinforcement learning problems of industrial scale are partially observable as it is generally too expensive or just impossible to collect all data determining the system's state space S. However, in the unlikely event that the system is fully observable, which means that $S = X$, equation 1.1 simplifies to

$$\mathbf{s}_{t+1} = g(\mathbf{s}_t, \mathbf{u}_t) \quad .$$

The corresponding reward function is denoted by $R : X \to \mathbb{R}$ (and respectively $R : S \to \mathbb{R}$ in the case of a fully observable problem). Hence it is based on the current observation and respectively state of the system.[2] Generally the particular immediate reward $R_t := R(x_t)$ is not of major interest as a corresponding action might lead to low rewards in the long run. Therefore, an accumulated reward over time with a possible discount factor $\gamma \in [0, 1]$ is regarded:

$$R := \sum_{t=1}^{\infty} \gamma^{t-1} R_t \tag{1.2}$$

Based on this, the underlying challenge for every high-dimensional and complex control task can be seen as a two step problem:

[1] Other formulations of controlled dynamical systems can be found in [27, 98].

[2] Other forms of reward functions, which e.g., also take the applied action into account, can be found in [89].

(i) System identification: The majority of technical and economical control systems cannot be described or represented by simple mathematical equations as most often the physical or economical interdependencies within the system are not (yet) fully explored or the system is simply only partially observable. Consequently, a model of the system is not a priori known. It rather has to be learnt out of the data collected during system operation. A model-building approach is therefore of avail as it allows to derive the future behaviour of the system given a certain control sequence. As the amount of available data is, due to time or cost restrictions, further generally limited, a data-efficient system identification is of importance. Moreover, system identification is the crucial part of solving a control problem as it serves as a basis for the second step, where the actual optimal control policy is learnt.

With regard to the open dynamical system (eq. 1.1), the related optimisation task consists of determining two functions $\bar{g} : \mathbb{R}^{\bar{J}} \times \mathbb{R}^K \rightarrow \mathbb{R}^{\bar{J}}$ and $\bar{h} : \mathbb{R}^{\bar{J}} \rightarrow \mathbb{R}^I$ (with $\bar{J} \in \mathbb{N}$), such that the error between the observable data, determined by the model, $\bar{\mathbf{x}}_t \in \mathbb{R}^I$, and the one of the real system, $\mathbf{x}_t$, is minimal for all available data patterns $t = 1, \ldots, T \in \mathbb{N}$:

$$\sum_{t=1}^{T} \|\bar{\mathbf{x}}_t - \mathbf{x}_t\|^2 \rightarrow \min_{\bar{g},\bar{h}} \tag{1.3}$$

This implies that the unknown system state $\mathbf{s}_t \in \mathbb{R}^J$ is modelled by the model's inner state $\bar{\mathbf{s}}_t \in \mathbb{R}^{\bar{J}}$, which might be of a different dimension, i.e. $J \neq \bar{J}$. Note that in this step there is no change in the applied control parameters $\mathbf{u}_t$. These are given as a part of the available data set and are used to learn their effects on the system's dynamics. In case that a model of the analysed system is known and can easily be described by a set of differential equations, one can directly refer to step (ii). However, throughout this thesis it is assumed that a model has to be determined out of an available data set.

(ii) Learning of the optimal control policy: The determination of the optimal control policy is based on the model identified in step (i). The objective is to find an optimal control policy, which maximises the future rewards R_t (eq. 1.2) of the RL problem. In the ideal case one can achieve a stable operating point of the system but mostly a continuous adjustment of the control parameters is required. In other words, the objective of step (ii) is the calculation of an optimal action selection or control policy, $\bar{\pi} : \mathbb{R}^{\bar{J}} \times \mathbb{R}^I \rightarrow \mathbb{R}^K$, which determines the model's next action $\bar{\mathbf{u}}_t \in \mathbb{R}^K$ based on the approximated inner state of the system $\bar{\mathbf{s}}_t \in \mathbb{R}^{\bar{J}}$ and the calculated latest

observation $\bar{\mathbf{x}}_t \in \mathbb{R}^I$,

$$\bar{\mathbf{u}}_t \quad = \quad \bar{\pi}(\bar{\mathbf{s}}_t, \bar{\mathbf{x}}_t)$$

under the consideration of the reward function R (eq. 1.2). This results in the following optimisation problem

$$\sum_{t=1}^{T} \gamma^{t-1} R_t \rightarrow \max_{\bar{\pi}} \quad . \tag{1.4}$$

1.2 Main Focus of the Thesis

The main objective of the thesis is the development of a new model-based reinforcement learning method on the basis of recurrent neural networks for a data-efficient solution to the described reinforcement learning problems (sec. 1.1). The approach focuses on a novel connection between reinforcement learning (RL) (chap. 2) and recurrent neural networks (RNN) (chap. 3) with the objective to solve the outlined two step problem (sec. 1.1).

RNN are used as they allow for the identification of dynamical systems in form of high-dimensional, non-linear state space models. Moreover, the architecture of RNN allows for a perfect modelling of the RL environment over a certain number of consecutive time steps. Also, they have shown to be very data-efficient. Therefore they are well suited for an application to the described extensive RL problems. To strengthen their utilisation it is proven that RNN possess a universal approximation ability and that they are well able to learn long-term dependencies. Also, some elaborated model-building approaches for RNN are presented. In this regard the thesis also provides important new theoretic results on RNN.

For the development of the aspired recurrent neural reinforcement learning approach the focus is on new neural network architectures instead of algorithms. In doing so the new methods not only learn from data but also integrate prior knowledge into the modelling in form of architectural concepts. This aims at supporting the efficient learning and mapping of the full environment of an RL problem. Furthermore, in contrast to most RL methods, the optimal policy is determined directly without making use of an evaluation respectively value function (sec. 2.3). This distinguishes the approach also from other works on RL with different kinds of recurrent networks (chap. 4).

However, the thesis does not aim at improving or even outperforming those existing approaches. It rather presents a new application of the described RNN (chap. 3) to RL (chap. 2) and shows that this combination can be successfully used to solve the aforesaid RL problems (sec. 1.1).

The new methods are tested on several standard benchmark problems, like different settings of the cart-pole (sec. 4.2) and mountain car problem (sec. 4.6). Still, from the application point of view, the main focus has been put on the control of gas turbine simulations as used in industry. Those mainly show the focused characteristics of RL problems, high-dimensionality, partial observability, continuous state and action spaces and a requirement for data-efficiency (sec. 2.5). They also motivated the generated work and underlying research.

Based on the mentioned characteristic figure 1.1 illustrates the main objective of the thesis and classifies it in terms of data efficiency and the ability to solve complex, i.e., high-dimensional, partially observable and with continuous state and action spaces, RL problems.

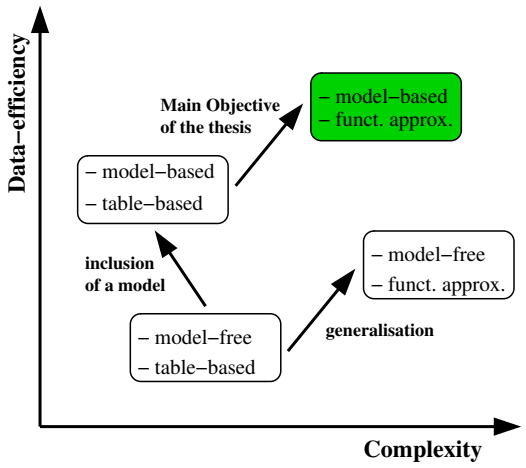


Figure 1.1: Main objective of the thesis classified in comparison to other principal classes of standard RL methods (sec. 2.4) with regard to data-efficiency and the ability to solve complex (high-dimensional, partially observable, continuous state and action spaces) RL problems. The purpose is the development of a model-based method on the basis of RNN for a data-efficient solution to complex RL problems.

1.3 Structure of the Thesis

Chapter 1 outlines the problem setting and states the main focus of the thesis.

Chapter 2 forms a brief introduction to reinforcement learning. The presentation focuses on basic aspects, which are relevant for the further course of the thesis. It enfolds a description of Markov decision processes and dynamic programming. Furthermore, standard reinforcement learning algorithms, which are

used as benchmarks in chapters 4 and 5, are introduced and analysed. Finally, the key problems and requirements for real-world applied reinforcement learning algorithms are discussed in more detail.

In chapter 3 recurrent neural networks and their capability to identify open dynamical systems are introduced and described. A proof for their universal approximation capability is given. Learning techniques and especially the capability to learn long-term dependencies are examined, analysed and discussed. Those aspects, long-term learning and universal approximation, form a basis and prerequirement for the application of the networks to RL and control problems. Additionally, a couple of modelling issues, which have been shown to be valuable in practise, are presented.

Chapter 4 presents the novel conjunctions between recurrent neural networks and reinforcement learning. As a first recurrent neural RL approach, only the ability of RNN to model and reconstruct the (approximately) Markovian state space of an RL problem is used. This is especially useful for partially observable RL problems. In this context a hybrid RNN approach is presented, where the RNN identifies the problem's dynamics (step (i)) and subsequent to that standard RL algorithms are applied on the networks inner state space to learn the optimal policy (step (ii)). On this basis the recurrent control neural network (RCNN) is introduced, which combines the two steps ((i) + (ii)) into one integrated neural network. Consequently, it not only learns the underlying dynamics but also determines the optimal policy. In an extended version, the idea of approximating the system's (minimal) Markovian state space is further incorporated. Thus, the extended RCNN embeds the presented hybrid RNN approach within one single neural network. Furthermore, the extended network calculates the changes in the control parameters instead of absolute values. Both aspects marked crucial improvements for controlling the gas turbine simulations (chap. 5). The two versions of RCNN and the idea of a state space reconstruction with RNN form the key contribution of this thesis. They have been filed for patents by Siemens AG.

In chapter 5 the novel recurrent neural RL approaches are tested on a real-world problem, the control of different kinds of gas turbine simulations. The problem is, within the limits of confidentiality, described and analysed. The new methods are applied and compared to standard controllers. It turns out that a significant increase in performance can be achieved.

Finally chapter 6 summarises the main results of the thesis.

”Knowledge must come through action; you can have no test which is not fanciful, save by trial.”

Sophocles (496 BC – 406 BC)

CHAPTER 2

Reinforcement Learning

Reinforcement learning (RL) combines the fields of dynamic programming [11] and supervised learning to develop powerful machine learning algorithms [32]. Besides its use for solving control problems, RL can be seen as “one of the only designs of value in understanding the human mind” [96]. It is an approach to learn an optimal behaviour, i.e. policy, in an unknown or at most partially known environment. Thereby, it is based on the idea of trial-and-error interactions with the dynamic environment [32, 42, 89]. The main elements of an RL or control problem (sec. 1.1) are shortly summarised in the following. Their interrelationship is also illustrated in figure 2.1:

(i) Environment: The environment corresponds to any kind of economical or technical system, e.g. stock market, revenue management, or a gas turbine. Its development is based on its history and the actions performed by the agent. For each interaction it sends a reward R_t to the agent, which serves as an evaluation criteria for the agent's action in the last system state. The state $\mathbf{s}_t$ of the environment can be discrete or continuous.

(ii) Agent: The agent represents the controller of the system. It can at least partially observe the system's state $\mathbf{s}_t$ by receiving observations $\mathbf{x}_t$. Using those, it interacts with the environment by performing actions $\mathbf{u}_t$ and in return retrieving rewards R_{t+1}, which it can use to improve its policy.

(iii) Actions: Actions influence the development of the environment. They generally represent a chosen change in or an absolute value of the control parameters allowed by the system. According to the problem setting those can be discrete or continuous. Here, a credit-assignment problem [54] has to be solved as the temporal dependencies are generally a priori unknown, i.e., some actions may not immediately change the system but may do so with a certain delay. Actions can be bounded or limited by the problem setting.

Examples for actions are the decision of buying or selling, adjustments of a temperature or of a steering wheel, or simply the application of a certain physical force to the environment.

(iv) Policy: The mapping from states of the environment to the action to be taken in this state is called a policy π. Following a policy forms a sequence of actions and reflects the learnt behaviour of the agent at a given time. For most applications one is interested in controlling a system over a certain time period. Therefore, instead of a one-step optimisation, one tries to determine an optimal policy with regard to a given overall objective, respectively reward function. According to Sutton and Barto "the policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behaviour" [89].

(v) Reward / cost function[1]**:** The reward function specifies the overall objective of the reinforcement learning problem. It depicts the immediate reward the agent receives for performing a certain action at a given system state. Consequently, it defines the desirability of an event for the agent. Generally the simple immediate reward is only of minor interest because high immediate rewards might lead to low ones in the future. Instead, one is usually interested in the (discounted) value of collected rewards in the long run.

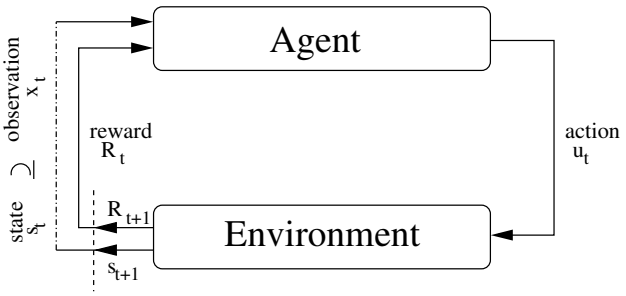


Figure 2.1: Basic idea of reinforcement learning: An agent iteratively interacts with an environment by carrying out an action $\mathbf{u}_t$ based on its observed state information $\mathbf{x}_t$, which can be smaller (partially observable) or equal (fully observable) to the environmental state $\mathbf{s}_t$. In return it retrieves a feedback in form of a reward R_{t+1}, which it uses to improve its policy π and thereby increase its future sum of rewards ($t = 1, \ldots, \infty$) [89]. The dashed line indicates the transition to the next time step.

[1]For simplification in the following it is mostly referred to reward functions. However, reward maximisation and cost minimisation can be used interchangeably.

2.1 Markov Decision Process

The mathematical basis for most theoretical RL problems is a Markov decision process (MDP), which describes the development of a (fully observable) controllable dynamical system (eq. 1.1). An MDP is basically defined by a tuple (S, U, Tr, R) with the following objects [18, 41, 65], whereby $t \in \mathbb{N}$ indicates the time step:

- a state space of the environment S,
- an action or control space U, with sets $U(\mathbf{s}_t)$ of available or allowed actions in state $\mathbf{s}_t \in S$,
- a deterministic or stochastic state-transition function $Tr(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{u}_t) : S \times U \times S \to [0, 1]$, which defines the probability for reaching state $\mathbf{s}_{t+1}$ being in state $\mathbf{s}_t$ and applying action $\mathbf{u}_t$ with $\mathbf{s}_t, \mathbf{s}_{t+1} \in S$ and $\mathbf{u}_t \in U(s_t)$,[2]
- a reward function $R_t := R(\mathbf{s}_t) : S \to \mathbb{R}$ denoting the immediate one-step reward for being in state $\mathbf{s}_t$.[3]

The relation of the different objects is described by a one-step transition: Given an open (controllable) dynamical system (eq. 1.1) with a state space S. Being in an arbitrary state $\mathbf{s}_t \in S$ at time step t, the agent chooses an action $\mathbf{u}_t \in U(\mathbf{s}_t)$. As a consequence the system evolves to the next state $\mathbf{s}_{t+1} \in S$ according to the transition function $Tr(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{u}_t)$. At the same time the agent receives the one-step reward $R(\mathbf{s}_{t+1})$ [18]. The generated sequence of states and actions is called a trajectory. Due to the assumed Markov property (defn. 2.1) the next state $\mathbf{s}_{t+1}$ hereby only depends on the current state $\mathbf{s}_t$ and the applied action $\mathbf{u}_t$. In other words, the Markov property states that the development of the system only depends on the last system state and the taken action [18, 50]. Consequently, it is independent of its history, i.e. the previous states and actions.

In mathematical terms the Markov property in discrete time is stated as follows, whereby $\mathbf{s}_0$ stands for an arbitrary starting state [50].

Definition 2.1. *Markov property: A discrete stochastic process $\mathbf{s}_t \in S$ with action $\mathbf{u}_t \in U$ and a transition function $Tr(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{u}_t)$ is called Markovian if for every $t \in \mathbb{N}$ it is*

$$Tr(\mathbf{s}_{t+1}|\mathbf{u}_t, \mathbf{s}_t, \mathbf{u}_{t-1}, \mathbf{s}_{t-1}, \ldots, \mathbf{u}_0, \mathbf{s}_0) = Tr(\mathbf{s}_{t+1}|\mathbf{u}_t, \mathbf{s}_t) \quad .$$

[2] In case of a continuous state space Tr is defined as a probability density function.

[3] Again, it is also possible to define the reward function such that it takes the applied action or even the full transition $(\mathbf{s}_t, \mathbf{u}_t, \mathbf{s}_{t+1})$ into account. An extension is trivial, but for simplicity throughout this thesis it is referred to the described form.

The actions can be chosen on the basis of a pre-defined (or learnt) decision rule, i.e. policy, or simply randomly. If the state and action spaces, S and U, are discrete, the MDP is called discrete. In the non-discrete case, one naturally assumes that they are measurable spaces endowed with σ-algebras $\mathcal{S}$ and $\mathcal{U}$ [18]. This comprises the case that S and U are continuous spaces in $\mathbb{R}^J$ and $\mathbb{R}^K$. The MDP is called deterministic if $Tr(\mathbf{s}_{t+1}|\mathbf{u}_t, \mathbf{s}_t)$ is deterministic and stochastic otherwise.

Following the illustration of basic reinforcement learning (fig. 2.1), figure 2.2 gives a (general) graphical representation of a Markov decision process. As the system is per definition fully observable, the agent's observation $\mathbf{x}_t$ is equal to the environmental state $\mathbf{s}_t$. Due to the Markov property (defn. 2.1) this contains all required information for the agent to determine its next action $\mathbf{u}_t$. Its decision rule and respectively policy is therefore a direct mapping from the observed state $\mathbf{x}_t(= \mathbf{s}_t)$ to the next action $\mathbf{u}_t$. The environment then evolves according to the transition function $Tr(\mathbf{s}_{t+1}|\mathbf{u}_t, \mathbf{s}_t)$.

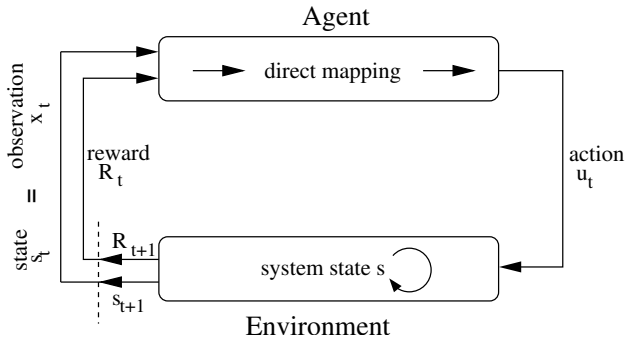


Figure 2.2: Graphical representation of a Markov decision process. As the system is per definition fully observable, it is $\mathbf{x}_t = \mathbf{s}_t$. Due to the Markov property the agent's decision making process is a direct mapping from the observed state to its action. Anew, the dashed line indicates the transition to the next time step.

2.2 Partially Observable Markov Decision Process

Partially observable Markov decision processes (POMDP) differ from MDP (sec. 2.1) in the fact that the state space S is not fully observable. This is generally the case in real-world applications, e.g. gas turbine control (chap. 5). The agent only receives an observation $\mathbf{x}_t \in X$ as an indicator for the actual state of the system, $\mathbf{s}_t \in S$. Hereby $\mathbf{x}_t$ is generally not Markovian. Formally a POMDP can

be described by a tuple (S, X, U, Tr, R), where in addition to MDP X represents the observation space, which can be a subspace of the state space S but might also include redundant information.

Figure 2.3 gives a (general) graphical representation of a partially observable Markov decision process. Unlike in MDP (fig. 2.2) the environmental state $\mathbf{s}_t$ is now only partially observable by the agent, which is depicted by the expression $\mathbf{x}_t \subset \mathbf{s}_t$. This implies that the agent has the additional task to approximate or reconstruct the environmental, Markovian state $\mathbf{s}_t$ out of its observations $\mathbf{x}_t$ to determine its next action $\mathbf{u}_t$. In other words, the agent has to build a model of the environment, which it uses as a basis for its decision making. Therefore also past time information about the system's development can be helpful.

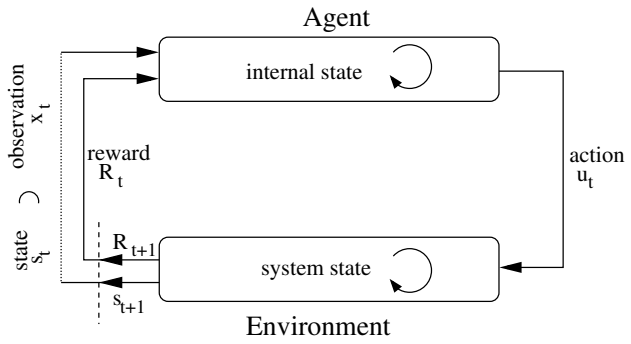


Figure 2.3: Graphical representation of a partially observable Markov decision process. As the system is partially observable, the agent only receives an observation $\mathbf{x}_t$ as an indicator for the system state $\mathbf{s}_t$ ($\mathbf{x}_t \subset \mathbf{s}_t$). Therefore, it builds up an internal state out of past time information to determine the next action $\mathbf{u}_t$. Again, the dashed line indicates the transition to the next time step.

As already pointed out, most real-world RL applications are partially observable (sec. 1.1). Therefore partial observability (sec. 2.5.2) is also in the main focus of this thesis. In chapter 4 new RNN based RL approaches are presented, which amongst others reconstruct the state space of a POMDP.

2.3 Dynamic Programming

The term dynamic programming (DP) refers to a group of algorithms, which can be used to solve multi-state decision processes with a perfect model of the environment, like MDP [89]. It is based on Bellman's principle of optimality, which can be formally written in the following form [6, 10]:

Theorem 2.1. ***Principle of Optimality***
Let $\{\mathbf{u}_0^*, \mathbf{u}_1^*, \mathbf{u}_2^*, \mathbf{u}_3^*, \ldots\}$ *be an action sequence resulting from an optimal policy* π^* *for the basic (fully observable) problem and assume that when using* π^* *a given state* $\mathbf{s}_t$ *occurs at time* t *with positive probability. Consider the sub-problem whereby one is at* $\mathbf{s}_t$ *at time* t *and wishes to maximise the "reward-to-go" from time* t *on with a discount factor* $\gamma \in [0, 1]$

$$\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(\mathbf{s}_{\tau+1})$$

Then the truncated action sequence $\{\mathbf{u}_t^*, \mathbf{u}_{t+1}^*, \mathbf{u}_{t+2}^*, \ldots\}$ *is optimal for this sub-problem.*

The explanation is quite intuitive. If the solution to the sub-problem was not optimal, the total reward of the problem could be further increased by switching to the optimal policy when being at state $\mathbf{s}_t$. Hence, π^* could not be optimal [10]. In return this implies that the optimal policy can be determined by solving step by step the respective "tail sub-problem", which is the basic principle of the dynamic programming algorithm [10].

Based on the principle of optimality DP operates on a so-called value function $V^\pi(\mathbf{s}_t)$, which represent the (expected) reward-to-go [10] for each system state $\mathbf{s}_t$ given a policy $\pi : S \to U$:

$$V^\pi(\mathbf{s}_t) = \mathbf{E}\left(\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(\mathbf{s}_{\tau+1})\right)$$

The DP algorithm, which is also called value iteration, aims at maximising the value function by proceeding a backward iteration ($k \in \mathbb{N}$)[10]:

$$V_{k+1}(\mathbf{s}_t) = \max_{\substack{\mathbf{u}_t \\ \in U(\mathbf{s}_t)}} \left(\sum_{\mathbf{s}_{t+1} \in S} Tr(\mathbf{s}_{t+1}|\mathbf{u}_t, \mathbf{s}_t)\left[R(\mathbf{s}_{t+1}) + \gamma V_k(\mathbf{s}_{t+1})\right]\right) \quad \forall t \qquad (2.1)$$

It has been shown to converge to the correct V^*, the value function of the optimal policy π^* [6, 9, 42].

The maximisation of V is done over the policy space because the dynamics of the state-action space is given by the underlying system respectively problem setting. Typically one takes an intermediate step and regards a so-called Q-function, which takes in addition to the value function also the chosen action $\mathbf{u}_t$ into account. This allows for an evaluation of every state-action pair instead of the states only. The Q-function is defined by

$$Q^\pi(\mathbf{s}_t, \mathbf{u}_t) = \sum_{\mathbf{s}_{t+1} \in S} Tr(\mathbf{s}_{t+1}|\mathbf{u}_t, \mathbf{s}_t)\left[R(\mathbf{s}_{t+1}) + \gamma Q^\pi(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1}))\right] \quad \forall t.$$

Furthermore, it is $V^{\pi}(\mathbf{s}_t) = Q^{\pi}(\mathbf{s}_t, \pi(\mathbf{s}_t)) \quad \forall t$.

Analogue to V^* the optimal Q-function is set as

$$\begin{aligned} Q^*(\mathbf{s}_t, \mathbf{u}_t) &:= \sum_{\mathbf{s}_{t+1} \in S} Tr(\mathbf{s}_{t+1} | \mathbf{u}_t, \mathbf{s}_t) \left[R(\mathbf{s}_{t+1}) + \gamma V^*(\mathbf{s}_{t+1}) \right] \\ &= \sum_{\mathbf{s}_{t+1} \in S} Tr(\mathbf{s}_{t+1} | \mathbf{u}_t, \mathbf{s}_t) \left[R(\mathbf{s}_{t+1}) + \gamma \max_{\substack{\mathbf{u}_{t+1} \\ \in U(\mathbf{s}_{t+1})}} Q^*(\mathbf{s}_{t+1}, \mathbf{u}_{t+1}) \right] \quad \forall t \end{aligned} \tag{2.2}$$

The latter (eq. 2.2) is called the Bellman optimality equation [6]. The optimal policy π^* is the one maximising the Q-function:

$$\pi^*(s_t) = \arg \max_{\substack{\mathbf{u}_t \\ \in U(\mathbf{s}_t)}} Q^*(\mathbf{s}_t, \mathbf{u}_t) \quad \forall t$$

A variation to value iteration (eq. 2.1) is a so-called policy iteration [42, 89], which directly takes the policy into account. Here, the value function is determined by doing a policy evaluation for a given policy π_i ($i, k \in \mathbb{N}$):

$$V_{k+1}^{\pi_i}(\mathbf{s}_t) = \sum_{\mathbf{s}_{t+1} \in S} Tr(\mathbf{s}_{t+1} | \mathbf{u}_t, \mathbf{s}_t) \left[R(\mathbf{s}_{t+1}) + \gamma V_k^{\pi_i}(\mathbf{s}_{t+1}) \right] \quad \forall t$$

In fact, this is simply the expected infinite discounted reward, which will be gained when following policy π_i. In practical applications the equation is iterated until $|V_{k+1}^{\pi_i}(\mathbf{s}_t) - V_k^{\pi_i}(\mathbf{s}_t)| < \varepsilon \quad \forall t$, with $\varepsilon > 0$. In a second step policy iteration determines whether this value could be improved by changing the immediate action taken. This results in the following policy update:

$$\pi_{i+1}(\mathbf{s}_t) = \arg \max_{\substack{\mathbf{u}_t \\ \in U(\mathbf{s}_t)}} \left(\sum_{\mathbf{s}_{t+1} \in S} Tr(\mathbf{s}_{t+1} | \mathbf{u}_t, \mathbf{s}_t) \left[R(\mathbf{s}_{t+1}) + \gamma V^{\pi_i}(\mathbf{s}_{t+1}) \right] \right) \quad \forall t$$

The two steps are iterated until the policy becomes stable, i.e. $\pi_i = \pi_{i+1}$.

Comparing value and policy iteration it has been shown in practice that the first is much faster per iteration but the latter needs fewer iterations. For both, justifications have been brought up, why they are better suited for large problems. Also modifications, especially with a focus on speed, have been developed [42].

Due to the requirement of a perfect model the standard DP algorithms are only of limited utility for extensive RL or control problems. Still, they serve as a basis for a couple of further developed methods, e.g. in combination with feedforward neural networks [11]. Moreover, they can been seen as the foundation of modern reinforcement learning [89].

2.4 Reinforcement Learning Methods

Several different RL methods have been developed over the last years. A good introduction can be found in the book of Sutton and Barto [89]. In the following a couple of algorithms are presented, which are either used as a benchmark in the experiments of chapters 4 and 5 or are required as a reference.

There are many different ways to classify RL methods. A principle distinction can be made between table-based and function approximation methods. Table-based methods store the value of each state-action combination within a table. As the size of the table is computationally limited, those methods are mainly applied to RL problems with a low-dimensional discrete state space. Examples are Q-learning (sec. 2.4.2) and adaptive heuristic critic (sec. 2.4.3). In contrast, function approximation methods learn a mapping from state-action pairs to their respective value. Those methods can be easier applied to higher dimensions and continuous state and action pairs. Examples for those are temporal difference (TD-) methods (sec. 2.4.1) on local basis functions [89] or with neural networks [91, 92] as well as neural fitted Q-iteration [68].

Another important distinction can be made between model-free and model-based algorithms. In short, model-free methods learn a controller without learning a model, i.e., without using the transition function Tr. They directly learn from the available data without making a detour by building a model. This makes them generally fast and also easy to implement. Model-based methods however learn a model first and then use it to derive a controller. This requires additional computation but makes them generally more data-efficient and hence better applicable to extensive real-world problems like a gas turbine (chap. 5), which can hardly be controlled directly out of the available data. TD-learning (sec. 2.4.1) and also Q-learning (sec. 2.4.2) for example are model-free whereas DP (sec. 2.3) and prioritised sweeping (PS) (sec. 2.4.4) are model-based.

2.4.1 Temporal Difference Learning

Temporal difference (TD) learning [88] can be seen as a combination of dynamic programming (sec. 2.3) and Monte-Carlo [52] ideas, as its methods bootstrap like DP and directly learn from raw experience without a model of the dynamics [89]. Hence, in contrast to standard DP (sec. 2.3), TD-learning is model-free. In fact, the iteration over the value function is done without any knowledge of the underlying dynamics, i.e. the transition function is not explicitly taken into account. TD-learning served as a basis for many further developed algorithms like Q-learning (sec. 2.4.2) or SARSA [89]. They all focus on improving the basic method for a certain problem class. Therefore standard TD-learning is rarely applied but the extended algorithms are of high value for solving RL problems.

The standard update rule for TD-learning is defined as

$$V(\mathbf{s}_t) \leftarrow V(\mathbf{s}_t) + \alpha[R(\mathbf{s}_{t+1}) + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)]$$

with a learning rate $\alpha \in [0, 1]$, which can either be fixed or variable [89].

TD-learning can either be implemented table-based or with function approximation [89, 91, 92].

2.4.2 Q-Learning

Q-learning [93] represents an off-policy TD-algorithm and is considered to be one of the "most important breakthroughs in RL" [89]. Off-policy means that in contrast to standard TD-learning (sec. 2.4.1) the algorithm iterates without the assumption of a specific policy. It rather assumes that every state-action pair is performed infinitely often, independent of a particular policy. In its original form Q-learning is table-based. The different Q-values are all stored in a table, which is updated during every iteration. As Q-learning is based on TD-learning it is also model-free, which means that no knowledge about the underlying dynamics is required. Its update rule is defined as follows:

$$Q(\mathbf{s}_t, \mathbf{u}_t) \leftarrow Q(\mathbf{s}_t, \mathbf{u}_t) + \alpha \left(R(\mathbf{s}_{t+1}) + \gamma \max_{\substack{\mathbf{u}_{t+1} \\ \in U(\mathbf{s}_{t+1})}} Q(\mathbf{s}_{t+1}, \mathbf{u}_{t+1}) - Q(\mathbf{s}_t, \mathbf{u}_t) \right)$$

At hitherto, the learning rate $\alpha \in [0, 1]$ can be fixed or variable, which mainly is a trade-off between efficiency and accuracy. Convergence can only be guaranteed with a decreasing α. Still, this is based on the assumption that the number of observations goes to infinity. Since one focus of this thesis is data-efficiency, this theoretic result is of minor interest. Hence, for the experiments in chapters 4 and 5 the learning rate is kept fixed.

Again, several extensions have been developed for Q-learning. Most important is the replacement of the table by function approximation. In connection with neural networks, especially neural fitted Q-iteration [68] has shown remarkable results. Here the Q-values are approximated by a feedforward neural network (eq. 3.1). In contrast to similar approaches past-time transition triples $(\mathbf{s}_t, \mathbf{u}_t, \mathbf{s}_{t+1})$ are stored and reused. The updates are done offline on the basis of a batch of transition triples, which are provided as inputs to the neural network. Thus, the method significantly improves the learning quality and data-efficiency.

2.4.3 Adaptive Heuristic Critic

The adaptive heuristic critic algorithm (AHC) [5, 72] is an adaptive and model-free version of policy iteration (sec. 2.3), in which the value function is computed

by TD-learning. Similar to this, the process of learning the policy is separated from learning the value function. Basically the algorithm consists of two components, a critic and a controller part. The controller determines the policy π by maximising the heuristic value function V^{π} determined by the critic. The critic however learns the value function V^{π}, given the policy determined by the controller. In contrast to policy iteration, this is mostly done simultaneously although only the alternating implementation has shown to converge under appropriate conditions [100].

The AHC belongs to the class of actor-critic methods [45, 89]. Those all apply the same principle by separating the learning of the value function and the one of the controller. Over the last years all sorts of algorithms have been applied for both components, two even with kinds of recurrent neural network [4, 62]. A discussion on those is given in chapter 4.

2.4.4 Prioritised Sweeping

Prioritised Sweeping (PS) [55] is a model-based approach, which aims for the advantages of both DP (sec. 2.3) and TD-learning (sec. 2.4.1). The algorithm is very close to DP respectively value iteration (eq. 2.1) with the important difference that a backup is only done for the values of those states whose estimated value is changing significantly. For this PS keeps a list of states, prioritised by the size of their changes. When the top state in the list is updated, the effect on each of its predecessors is computed. If the effect is greater than some small threshold, the pair is inserted in the list or updated with the new priority. In this way the effects of changes are efficiently propagated backwards until some kind of convergence [89]. In doing so, the algorithm needs less computation and is hence much faster than standard DP. This is important when the number of states is increasing. Still, PS is generally implemented table-based, which limits its use for high-dimensional, real-world applications, e.g. control of a gas turbine (chap. 5).

2.4.5 Policy Gradient Methods

Policy gradient methods [90] represent the policy itself by a function approximator, which is independent of the value function and only updated according to the gradient information of the expected reward with respect to the policy parameters. By contrast to actor-critic methods (sec. 2.4.3), policy gradient methods not even make use of the value function. They directly search in the policy space. This resolves the disadvantage that a small shift of the value function can result in a decisive change of the policy. In contrast to value function approaches, policy gradient methods circumvent those discontinuities by learning the policy directly. Furthermore, they are known to be robust and to behave well in partially

observable domains [45, 64, 90]. Still, they usually base on Monte-Carlo estimations of the discounted future rewards [89], which implies that they are hardly data-efficient. Latest results and examples for different kinds of policy gradient methods can be found in [59, 70]. Besides, a conjunction of actor-critic and policy gradient methods forms the natural actor-critic method [60].

Also the in chapter 4 developed recurrent control neural network can be classified as a form of policy gradient method. A deeper discussion on this is given in section 4.4.

2.5 Classification of the Regarded RL Problems

There is a huge variety of RL-problems, depending on the respective setting and the underlying objective. In the following the regarded problem class of the thesis is described in detail and it is pointed out where the existing methods (sec. 2.4) have drawbacks or even fail to produce satisfactory results. Practical applications, like the control of a gas turbine, mainly show four important characteristics: High-dimensionality, partial-observability, continuous state and action spaces and a limited amount of training data, which in turn requires data-efficiency.

2.5.1 High-Dimensionality

In practical applications high-dimensionality usually comes out of the large amount of parameters, which influence the (extensive) dynamics of the regarded RL-problem in one way or the other. Reducing the amount by e.g. feature selection might result in a loss of important information. In addition, as the amount of data is often limited (sec. 2.5.4), it will be of advantage to take into account most of the available information concerning states and actions. For that reason RL methods, which are able to deal with high dimensions and which are easily scalable, are required. This basically excludes all table-based methods, as those are not representable in large dimensions. Rather an accurate and efficient function approximation is needed.

2.5.2 Partial-Observability

In plant control often several hundreds of state space variables (sec. 2.5.1) are measured. Still, those are mostly insufficient to fully describe the system behaviour. In other words, real-world applications are generally partially observable because it is either impossible or too expensive to observe all corresponding information. In those cases an optimal system identification of the underlying dynamics is beneficial. Hence, model-based approaches are of advantage as they first

reconstruct the system's dynamics out of the observed data. However, the quality of the model is crucial, as otherwise every policy learnt will be sub-optimal.

2.5.3 Continuous State and Action Spaces

Discrete state and action spaces, whereon most RL-methods focus, are rare in real-world problems. In most cases both are continuous. Table-based methods can handle those only with an appropriate discretisation, which tampers the problem. An appropriate function approximation is therefore of avail.

2.5.4 Data-Efficiency

In real-world applications the amount of available data is generally limited. Consequently, data-efficiency and generalisation capability are important requirements. It is hence of advantage to take the observed trajectories fully into account and to use them as efficient as possible. So far, most approaches with a focus on data-efficiency like least-squares policy-iteration [46] have the disadvantage of not being directly applicable in partially observable environments. They also possess practical limitations while applying them on continuous action spaces.

Classifying the presented RL methods (sec. 2.4) analogue to figure 1.1 according to the four characteristics one achieves the following graphical representation:

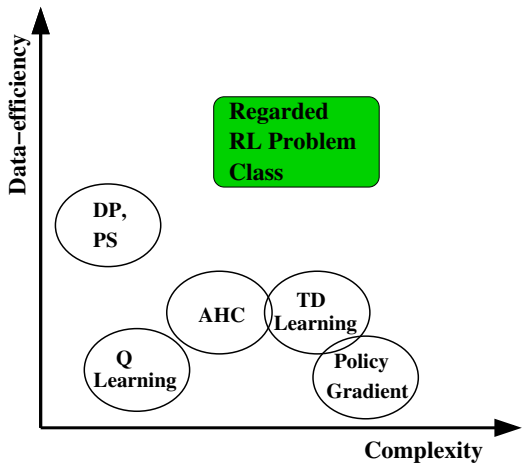


Figure 2.4: Classification of the presented RL methods according to the four characteristics of the regarded RL problem class. For simplicity high dimensionality, partial observability and continuous state and action spaces are again clustered as complex.

”The art of model-building is the exclusion of real but irrelevant parts of the problem, and entails hazards for the builder and the reader. The builder may leave out something genuinely relevant; the reader, armed with too sophisticated an experimental probe or too accurate a computation, may take literally a schematised model whose main aim is to be a demonstration of possibility.”

P. W. Anderson (from Nobel acceptance speech, 1977)

CHAPTER 3

System Identification with Recurrent Neural Networks

System identification comprehends the process of learning a model out of an available amount of observed system data. Thereby it is of importance to capture the main developments of the underlying system and hence to be able to generalise, i.e. to explain the system evolvement beyond the observed data region. For this, one needs amongst others to be able to build up memory, to learn long-term dependencies, capture non-linearities and deal with high dimensions.

Basic time delay recurrent neural networks were already applied to system identification in 1990 by Elman [17]. Similar types of recurrent networks were developed in parallel by Williams and Zipser [101] or Giles et al. [23]. These networks marked a starting point in system identification with recurrent neural network research, as they seemed to offer the mentioned abilities. However, a couple of problems, like the learning of long-term dependencies or the efficient training of the networks, still had to be solved. For this reason various kinds of recurrent networks have been developed, by what today the expression ”recurrent neural network” is not clearly defined in literature. Although, they are all more or less based on neural network techniques, their structure and functioning differs substantially.

The recurrent neural networks (RNN) used in this thesis are in state space model form. Their structure is based on the principle of finite unfolding in time and a shared weight extension of the backpropagation algorithm. The focus is on the mathematical modelling and the development of architectures instead of new algorithms. However, Zimmermann [103, 111] could show that a network architecture automatically implies the use of an adjoint solution algorithm for the respective parameter identification problem. This correspondence between architecture and equations holds for simple as well as extensive network architectures [106]. The application of backpropagation further allows for an easy extension

of the networks, as the algorithm is local in the sense that all necessary information is calculated at the affected place, which implies that no global information storage is required. This has the advantage that training becomes more efficient and that the networks can be adapted according to the regarded problem setting. In doing so, one not only learns from data but can also integrate prior knowledge into the modelling in form of architectural concepts. Moreover, in contrast to linear function approximators on local basis functions or with fixed local properties, the proposed RNN have the advantage that by using global, sigmoidal activation functions, they are well able to cope with higher dimensions. Thus, they are suited to break the curse of dimensionality [7]. It is further shown that they are universal approximators and that they are well able to learn long-term dependencies. These aspects are particularly important for the new connection with reinforcement learning (chap. 4). So far, the outlined RNN architectures have been mainly used for forecasting of e.g. foreign exchange rates or energy prices [104, 107, 110]. However, one can profit from the developments and experiences in this field for their application to RL problems (sec. 1.1).

Other recurrent networks, which have been developed with the particular intention to overcome the problems of basic time-delay recurrent neural networks [17, 23, 101] are for example echo-state [39, 40] and long short-term memory (LSTM) networks [22, 37]. Echo-state networks [39, 40] are constructed like feedforward neural networks, whereby the neurons in the hidden layer have recurrent but fixed connections. The only part that is learnt is a linear connection between hidden and output layer. This allows for a fast training, but leaves out the possibility to learn non-linearities, i.e., to adapt the features defined by the hidden layer. The input data is simply increased to a higher dimensional recurrent hidden layer, wherefrom the required information is taken to explain the desired output. Therefore, echo state networks can also be seen as a special form of linear feature selection. LSTM networks [22, 37] have again a completely different structure. Those networks possess so-called memory cells, which act as an additional layer to the network and store the essential inter-temporal information. They have been developed with a special focus on long-term learning, like in speech recognition [81]. However, despite their success in certain problem classes, in comparison to RNN they are not able to integrate prior knowledge into the modelling and to adapt the network accordingly. An overview of further developments on recurrent neural networks can be found in the books of Haykin [33, 34], Kolen and Kremer [44], Medsker and Jain [51], and Soofi and Cao [85].

In the following the outlined RNN are presented in detail. In preparation for this, first feedforward neural networks are briefly described (sec. 3.1). This is necessary as they serve as a basis for RNN. Besides, they are used to model the policy in the developed recurrent control neural network (sec. 4.4). In section 3.2 RNN are introduced. Subsequent to that it is proven that RNN are universal

approximators in the sense that they can approximate any open dynamical system (sec. 3.3). In section 3.4 the learning of RNN is described and analysed. Here, it is especially shown that RNN are well able to learn long-term dependencies. As already pointed out, being able to universally approximate and to learn long-term dependencies are key requirements for the aspired connection of RNN with reinforcement learning (chap. 4). Finally in section 3.5 a number of practical issues are discussed, which have shown to be very useful for RNN modelling.

3.1 Feedforward Neural Networks

Multi-layer feedforward neural networks (FFNN), also called multi-layer perceptrons (MLP), basically consist of an input, a number of hidden, and an output layer. Between the different layers there is a weighted forward directed information flow from the input layer over the hidden to the output layer. There are no backward connections between or within the layers [33], which means that the output of one layer cannot be used as an input to a previous or the same one. FFNN therefore mainly serve for a static information processing or respectively pattern matching.

Consider a three-layered FFNN, which is per definition composed of an input, one hidden and an output layer. Let $\mathbf{x}_1, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_T$, with $\mathbf{x}_t \in \mathbb{R}^I$ and $I \in \mathbb{N}$, and $\mathbf{y}_1, \ldots, \mathbf{y}_t, \ldots, \mathbf{y}_T$, with $\mathbf{y}_t \in \mathbb{R}^N$ and $N \in \mathbb{N}$, be the measured or given input and respectively output data. Again, $T \in \mathbb{N}$ denotes the number of regarded data patterns. Further, let $\bar{J} \in \mathbb{N}$ be the dimension of the single hidden layer.

A three-layered FFNN can then be represented by the following equation

$$\bar{\mathbf{y}}_t = V \cdot f(W \cdot \mathbf{x}_t - \theta) \quad \forall t = 1, \ldots, T \quad , \tag{3.1}$$

where $W = (w_{ji})_{\substack{j=1,\ldots,\bar{J} \\ i=1,\ldots,I}} \in \mathbb{R}^{\bar{J} \times I}$ and $V = (v_{nj})_{\substack{n=1,\ldots,N \\ j=1,\ldots,\bar{J}}} \in \mathbb{R}^{N \times \bar{J}}$ are the weight matrices between input and hidden and respectively hidden and output layer, $\theta \in \mathbb{R}^{\bar{J}}$ is a bias, which handles offsets in the inputs, and $f(\cdot) : \mathbb{R}^{\bar{J}} \to \mathbb{R}^{\bar{J}}$ an arbitrary, but generally sigmoid (defn. 3.3), (non-linear) activation function.

Here, in short, the information flow proceeds as follows: The input vector $\mathbf{x}_t$ is transferred from the input layer to the hidden layer by a multiplication with the weight matrix W. There, the neurons are mapped by the activation function $f(\cdot)$. Finally the network's output $\bar{\mathbf{y}}_t \in \mathbb{R}^N$ is calculated according to the weight matrix V [33].

The network is trained by comparing its outputs $\bar{\mathbf{y}}_t$ with the given target values $\mathbf{y}_t$ and adapting the weights such that the error, i.e. the deviation between output and target, becomes minimal (sec. 3.4). Therefore, neural networks in general belong to the class of supervised learning.

Remark 3.1. *The hyperbolic tangent,* $\tanh$*, has shown to be very suitable as an activation function for most networks and applications due to its characteristics like non-linearity, differentiability and its codomain* $(-1, 1)$*, which allows positive and negative forward (and backward) flows within the network. The latter corresponds to positive and negative influences of the weights on the next network layer. Still, any other (sigmoid) activation function (def. 3.3) could be applied.*

In any case it is important to note that in the context of artificial neural networks the computation of the activation function $f(\cdot) : \mathbb{R}^{\bar{J}} \to \mathbb{R}^{\bar{J}}$ *is defined component-wise, i.e.,*

$$f(W\mathbf{x}_t - \theta) := \begin{pmatrix} f(W_1 \cdot \mathbf{x}_t - \theta_1) \\ \vdots \\ f(W_j \cdot \mathbf{x}_t - \theta_j) \\ \vdots \\ f(W_{\bar{J}} \cdot \mathbf{x}_t - \theta_{\bar{J}}) \end{pmatrix} \tag{3.2}$$

where W_j $(j = 1, \ldots, \bar{J})$ *denotes the* $j - th$ *row of the matrix* W*.*

It has been proven that a three-layered feedforward neural network (eq. 3.1) is already able to approximate any measurable function on a compact domain with an arbitrary accuracy [15, 19, 38]. The main steps of the respective proof of Hornik, Stinchcombe and White [38] are given in section 3.3.1.

However, a disadvantage of FFNN is their lack of recurrence, which limits the incorporation of inter-temporal dependencies. By construction they can only perform a pattern matching from inputs to outputs, which makes their application to (non-Markovian) dynamical systems questionable.

3.2 Recurrent Neural Networks

System identification with recurrent neural networks (RNN) originally refers to open dynamical systems (fig. 3.1), which are analogue to equation 1.1 described as a set of equations, consisting of a state transition and an output equation:[1]

$$\begin{aligned} \mathbf{s}_{t+1} &= g(\mathbf{s}_t, \mathbf{x}_t) \qquad \text{state transition} \\ \mathbf{y}_t &= h(\mathbf{s}_t) \qquad \text{output equation} \end{aligned} \tag{3.3}$$

where $g : \mathbb{R}^J \times \mathbb{R}^I \to \mathbb{R}^J$, with $J \in \mathbb{N}$, is a measurable and $h : \mathbb{R}^J \to \mathbb{R}^N$ a continuous function, $\mathbf{x}_t \in \mathbb{R}^I$ represents the external inputs, $\mathbf{s}_t \in \mathbb{R}^J$ the inner states and $\mathbf{y}_t \in \mathbb{R}^N$ the output of the system [79]. The state transition is a mapping

[1] Alternative descriptions can be found in [79].

from the internal hidden state of the system $\mathbf{s}_t$ and the external inputs $\mathbf{x}_t$ to the next state $\mathbf{s}_{t+1}$. The output equation computes the observable output $\mathbf{y}_t$ out of the current state $\mathbf{s}_t$ [33, 109].

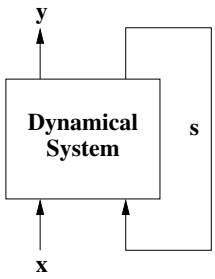


Figure 3.1: Open dynamical system with input $\mathbf{x}$, hidden state $\mathbf{s}$ and output $\mathbf{y}$.

The system can be seen as a partially observable autoregressive dynamic state transition $\mathbf{s}_t \rightarrow \mathbf{s}_{t+1}$ that is also driven by external forces $\mathbf{x}_t$. Without the external inputs the system is called an autonomous system [33, 49]. However, most real-world systems are driven by a superposition of an autonomous development and external influences.

The task of identifying the open dynamical system of equation 3.3 can be stated as the problem of finding (parametrised) functions $\bar{g}$ and $\bar{h}$ such that a distance measurement (eq. 3.4) between the observed data $\mathbf{y}_t$ and the computed data $\bar{\mathbf{y}}_t$ of the model is minimal:[2]

$$\sum_{t=1}^{T} \|\bar{\mathbf{y}}_t - \mathbf{y}_t\|^2 \rightarrow \min_{\bar{g},\bar{h}} \tag{3.4}$$

If one makes the assumption that the state transition does not depend on $\mathbf{s}_t$, i.e., $\mathbf{y}_t = h(\mathbf{s}_t) = h(g(\mathbf{x}_{t-1}))$, one is back in the framework of FFNN (sec. 3.1). However, the inclusion of the internal hidden dynamics makes the modelling task much harder, because it allows varying inter-temporal dependencies. Theoretically, in the recurrent framework an event in state $\mathbf{s}_{t+1}$ is explained by a superposition of external inputs $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots$ from all the previous time steps [33].

The identification task of equations 3.3 and 3.4 can be easily modelled by a recurrent neural network. Again, let therefore I, $\bar{J}$, and $N \in \mathbb{N}$ denote respectively the number of input, hidden and output neurons. For discrete time the basic RNN is depicted as follows [33, 109]:

$$\begin{aligned} \bar{\mathbf{s}}_{t+1} &= f(A\bar{\mathbf{s}}_t + B\mathbf{x}_t - \theta) && \text{state transition} \\ \bar{\mathbf{y}}_t &= C\bar{\mathbf{s}}_t && \text{output equation} \end{aligned} \tag{3.5}$$

[2] For other error functions see [57].

Here, the (non-linear) state transition equation $\bar{\mathbf{s}}_{t+1} \in \mathbb{R}^{\bar{J}}$ ($t = 1, \ldots, T$ where $T \in \mathbb{N}$ is the number of available patterns) is a non-linear combination of the previous state $\bar{\mathbf{s}}_t \in \mathbb{R}^{\bar{J}}$ and the external influences $\mathbf{x}_t \in \mathbb{R}^I$ using weight matrices $A \in \mathbb{R}^{\bar{J} \times \bar{J}}$ and $B \in \mathbb{R}^{\bar{J} \times I}$, and a bias $\theta \in \mathbb{R}^{\bar{J}}$, which handles offsets in the input variables $\mathbf{x}_t \in \mathbb{R}^I$. Note that the (non-linear) activation function f is applied component-wise (eq. 3.2). The network output $\bar{\mathbf{y}}_t \in \mathbb{R}^N$ is computed from the present state $\bar{\mathbf{s}}_t \in \mathbb{R}^{\bar{J}}$ employing matrix $C \in \mathbb{R}^{N \times \bar{J}}$. It is therefore a non-linear composition applying the transformations A, B, and C. Note here that the state space of the RNN $\bar{\mathbf{s}}_t \in \mathbb{R}^{\bar{J}}$ (eq. 3.5) generally does not have the same dimension as the one of the original open dynamical system $\mathbf{s}_t \in \mathbb{R}^J$ (eq. 3.3), i.e., in most cases it is $\bar{J} \neq J$. This basically depends on the system's complexity and the desired accuracy.

Training the RNN of equation (3.5) is equivalent to the described system identification (eq. 3.4) by specifying the functions $\bar{g}$ and $\bar{h}$ as a recurrent neural network with weight matrices A, B, and C and a bias vector θ. In doing so, the system identification task of equation 3.4 is transformed into a parameter optimisation problem:

$$\sum_{t=1}^{T} \|\bar{\mathbf{y}}_t - \mathbf{y}_t\|^2 \rightarrow \min_{A,B,C,\theta} \tag{3.6}$$

In section 3.3.2 it is proven that RNN (eq. 3.5) are universal approximators, as they can approximate any open dynamical system (eq. 3.3) with an arbitrary accuracy.

3.2.1 Finite Unfolding in Time

The parameter optimisation problem of equation 3.6 can be solved by finite unfolding in time using shared weight matrices A, B, and C [33, 71]. Shared weights share the same memory for storing their weights, i.e., the weight values are the same at each time step of the unfolding $\tau \in \{1, \ldots, T\}$ and for every pattern t [33, 71]. This guarantees that the dynamics stays the same in every time step. A major advantage of RNN written in form of a state space model (eq. 3.5) is the explicit correspondence between equations and architecture. It is easy to see that by using unfolding in time the set of equations 3.5 can be directly transferred into a spatial neural network architecture (fig. 3.2) [33, 71]. Here, circles indicate the different clusters, whereat the hidden layer $\bar{\mathbf{s}}_\tau$ also includes the (non-linear) transition function. Connections stand for the addition of the neurons of one layer (or the bias) to another one, respectively multiplied by one of the weight matrices.

The recurrence of the system is approximated with a finite unfolding, which truncates after a certain number of time steps $m_- \in \mathbb{N}$. Hereby, the determination

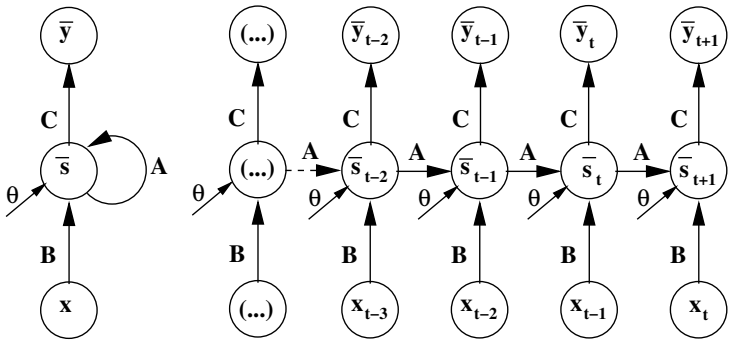


Figure 3.2: RNN with finite unfolding in time using shared weight matrices A, B and C: The recurrent network on the left is transferred to the spatial architecture on the right. Here, circles represent the different clusters, whereas connections stand for the addition of one layer (or the bias) to another one, respectively multiplied by one of the weight matrices. The dashed connection and (. . .) indicate that the network can be finitely further unfolded into the past.

of the correct amount of past time information needed to predict $\mathbf{y}_{t+1}$ can be determined by a simple heuristic. Since the outputs are explained by more and more external information, the error of the outputs is decreasing with each additional time step from left to right until a minimum error is achieved. This saturation level indicates the maximum number of time steps m_-, which contribute relevant information for modelling the present time state [109].

The unfolded RNN shown in figure 3.2 (right) can be trained with a shared weights extension of the standard backpropagation algorithm (sec. 3.4.1). Due to unfolding in time in comparison to equation 3.6 the corresponding optimisation problem is only slightly altered into:

$$\sum_{t=m_-}^{T} \sum_{\tau=t-m_-}^{t+1} \left\| \bar{\mathbf{y}}_\tau - \mathbf{y}_\tau \right\|^2 \rightarrow \min_{A,B,C,\theta} \tag{3.7}$$

In contrast to typical FFNN (sec. 3.1), RNN are able to explicitly build up memory. This allows the identification of inter-temporal dependencies. Furthermore, RNN contain less free parameters. In a FFNN an expansion of the delay structure automatically increases the number of weights (fig. 3.3, left). In the recurrent formulation, the shared matrices A, B, and C are reused when more delayed input information from the past is needed (fig. 3.3, right).

Additionally, if weights are shared more often, more gradient information is available for learning due to the numerous error flows when using backpropagation (sec. 3.4.1). As a consequence, potential overfitting is not as dangerous in

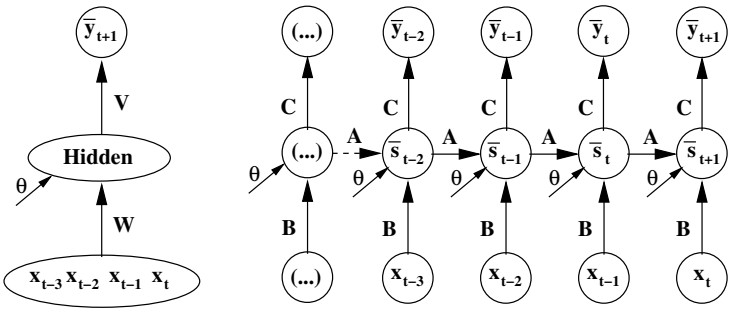


Figure 3.3: Comparison of feedforward and recurrent neural networks: An additional time step leads in the feedforward framework (left) to a higher dimension of the input layer, whereas the number of free parameters remains constant in RNN (right) due to the use of shared weights.

recurrent as in feedforward networks, which in turn implies that RNN are better able to generalise. Moreover, due to the inclusion of temporal structure in the network architecture, RNN are applicable to tasks where only a small training set is available [109]. This qualifies their application for data-efficient RL.

3.2.2 Overshooting

In its simplest form RNN unfolded in time only provide a one-step prediction of the variable of interest, $\bar{\mathbf{y}}_{t+1}$ (fig. 3.2). Especially with regard to RL but also for other decision support systems or simply multi-step forecasting this is generally insufficient. Most often one wants to evaluate the system's performance over a certain period of time and hence needs a sequence of forecasts as an output. An obvious generalisation of the network in figure 3.2 is therefore the extension of the autonomous recurrence coded in matrix A, i.e., matrices A and C are further iterated into future direction $t+1, t+2, \ldots$ [109]. The number of autonomous iterations into the future, which is defined with $m_+ \in \mathbb{N}$, most often depends on the required forecast horizon of the application, but can also be determined analytically by regarding the error flow. The later can be done analogue to the determination of the number of optimal unfolding steps into the past m_- (sec. 3.2.1) [109]. Note again that overshooting does not add new parameters, since the shared weight matrices A and C are reused.

Altogether an RNN with overshooting can be described by the following set of equations:

$$\begin{aligned} \bar{\mathbf{s}}_{\tau+1} &= f(A\bar{\mathbf{s}}_\tau + B\mathbf{x}_\tau - \theta) \quad \forall \tau \leq t \\ \bar{\mathbf{s}}_{\tau+1} &= f(A\bar{\mathbf{s}}_\tau - \theta) \quad \forall \tau > t \\ \bar{\mathbf{y}}_\tau &= C\bar{\mathbf{s}}_\tau \end{aligned} \tag{3.8}$$

Figure 3.4 depicts the corresponding network architecture. Here, the dotted connections indicate that the network can be (finitely) further unfolded into past and future.

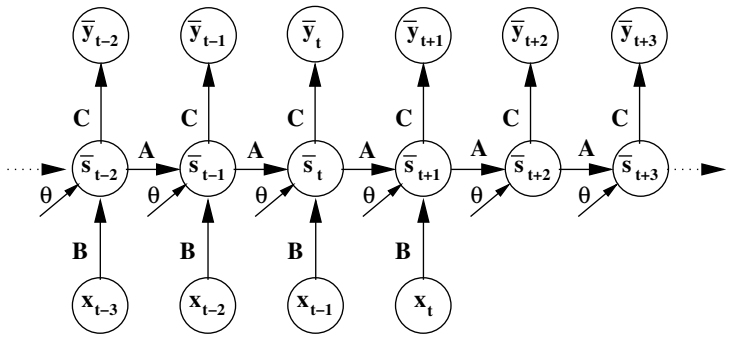


Figure 3.4: Overshooting extends the autonomous part of the dynamics into future direction.

The most important property of the overshooting network (fig. 3.4) is the concatenation of an input-driven system and an autonomous system. One may argue that the unfolding-in-time network (fig. 3.2) already consists of recurrent functions and that this recurrent structure has the same modelling characteristics as the overshooting network. This is not the case as the learning algorithm leads to different models for each of the architectures. Learning with backpropagation usually tries to model the relationship between the most recent inputs and the latest output because the fastest adaptation takes place in the shortest path between input and output [35]. Thus, the learning of $\mathbf{y}_{t+1}$ mainly focuses on $\mathbf{x}_t$. Only later in the training process learning will also extract useful information from input vectors $\mathbf{x}_\tau$ ($t - m_- \leq \tau < t$), which are more distant from the output. As a consequence, the simple RNN unfolded in time (fig. 3.2, right) tries to rely as much as possible on the part of the dynamics that is driven by the most recent inputs $\mathbf{x}_t, \ldots, \mathbf{x}_{t-k}$ with $k < m_-$. In contrast, the overshooting network (fig. 3.4) forces learning to focus on modelling the autonomous dynamics of the system, i.e., it supports the extraction of useful information from input vectors that are more distant to the output [109]. In extension to equation 3.7 the complete optimisation problem for an RNN unfolded in time and with overshooting is

$$\sum_{t=m_-}^{T-m_+} \sum_{\tau=t-m_-}^{t+m_+} \|\bar{\mathbf{y}}_\tau - \mathbf{y}_\tau\|^2 \rightarrow \min_{A,B,C,\theta} \tag{3.9}$$

In summary, overshooting generates additional valuable forecast information about the analysed dynamical system and stabilises learning. It also further enhances the generalisation ability of the RNN.

3.2.3 Dynamical Consistency

RNN with overshooting have the problem of unavailable external information $\mathbf{x}_\tau$ in the overshooting part $(\tau > t)$ of the network. Those are simply set to zero. This might form a gap in the input information between past and future part of the network, if it doesn't correspond to the expected value of the inputs. Even then neglecting the missing influences of future inputs is equivalent to the assumption that the environment of the dynamics stays constant or respectively that the external influences are not significantly changing, when the network is iterated into future direction. In other words the networks are not dynamically consistent [106]. Considering external variables with a high impact on the dynamics of interest like control parameters, this becomes questionable. Especially in long-term forecasts, it might lead to bad generalisation abilities.

RNN with dynamically consistent overshooting solve this problem by forecasting not only the variables of interest $\mathbf{y}_\tau$ but all environmental data $\mathbf{x}_\tau$ and using the network's own predictions for those, $\bar{\mathbf{x}}_\tau$, as a replacement for the unknown future inputs $\mathbf{x}_\tau$ $(\tau > t)$. In doing so the expected future development of the environment gets integrated into the modelling and the networks become dynamically consistent. As a side effect this allows for an integrated modelling of the dynamics of interest. Analogue to equations 3.8 and 3.9 RNN with dynamically consistent overshooting can be described by the following set of equations (eq. 3.10). Here, for $\tau \leq t$ the state transition uses the available external information $\mathbf{x}_\tau$ as inputs, whereas in the future part $(\tau > t)$ this gets replaced by the network's own predictions $\bar{\mathbf{x}}_\tau$.

$$\begin{aligned} \bar{\mathbf{s}}_{\tau+1} &= f(A\bar{\mathbf{s}}_\tau + B\mathbf{x}_\tau - \theta) \quad \forall \tau \leq t \\ \bar{\mathbf{s}}_{\tau+1} &= f(A\bar{\mathbf{s}}_\tau + B\bar{\mathbf{x}}_\tau - \theta) \quad \forall \tau > t \\ \bar{\mathbf{x}}_\tau &= C\bar{\mathbf{s}}_\tau \\ \sum_{t=m_-}^{T-m_+} \sum_{\tau=t-m_-}^{t+m_+} \|\bar{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2 &\rightarrow \min_{A,B,C,\theta} \end{aligned} \tag{3.10}$$

Again, it can be easily represented in an architectural form (fig. 3.5), where in the overshooting part $(\tau > t)$ of the network the (dashed) connections between the outputs $\bar{\mathbf{x}}_\tau$ and the states $\bar{\mathbf{s}}_\tau$ provide dynamical consistency. Again, the dotted connections indicate that the network can be (finitely) further unfolded into past and future.

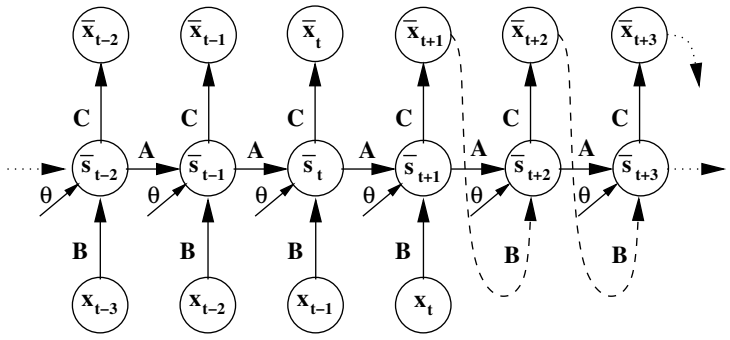


Figure 3.5: RNN with dynamically consistent overshooting. The network now forecasts all environmental data $\mathbf{x}_\tau$. Thus, the dashed connections between output and hidden layer provide dynamical consistency.

3.3 Universal Approximation

In 1989 Hornik, Stinchcombe, and White [38] proved that any Borel-measurable function on a compact domain can be approximated by a three-layered neural feedforward network (sec. 3.1) with an arbitrary accuracy. In the same year Cybenko [15] and Funahashi [19] found similar results, each with different methods. Whereas the proof of Hornik, Stinchcombe, and White [38] is based on the Stone-Weierstrass theorem [86, 94], Cybenko [15] makes in principle use of the Hahn-Banach and Riesz theorem. Funahashi [19] mainly applies the Irie-Miyake and the Kolmogorov-Arnold-Sprecher theorem.

Some work has already been done on the capability of RNN to approximate measurable functions, e.g. [30]. In the following it is proven that RNN (sec. 3.2) are able to approximate any open dynamical system with an arbitrary accuracy. The proof is based on the work of Hornik, Stinchcombe and White [38]. Therefore their definitions and main results are recalled in section 3.3.1.

3.3.1 Universal Approximation by FFNN

Definition 3.1. *Let* A^I *with* $I \in \mathbb{N}$ *be the set of all affine mappings* $A(\mathbf{x}) = \mathbf{w}\cdot\mathbf{x}-\theta$ *from* $\mathbb{R}^I$ *to* $\mathbb{R}$ *with* $\mathbf{w}, \mathbf{x} \in \mathbb{R}^I$ *and* $\theta \in \mathbb{R}$. *'$\cdot$' denotes the scalar product.*

Transferred to neural networks $\mathbf{x}$ corresponds to the input, $\mathbf{w}$ to the network weights and θ to the bias.

Definition 3.2. *For any (Borel-)measurable function* $f(\cdot) : \mathbb{R} \to \mathbb{R}$ *and* $I \in \mathbb{N}$ *be*

$\sum^I(f)$ *the class of functions*

$$\{NN: \ \mathbb{R}^I \to \mathbb{R} : NN(\mathbf{x}) = \textstyle\sum_{j=1}^{\bar{J}} v_j f(A_j(\mathbf{x})) \\ \mathbf{x} \in \mathbb{R}^I, v_j \in \mathbb{R}, A_j \in \mathrm{A}^I, \bar{J} \in \mathbb{N}\}.$$

Here NN stands for a three-layered feedforward neural network (sec. 3.1) with I input-neurons, $\bar{J}$ hidden-neurons and one output-neuron. v_j denotes the weights between hidden- and output-neurons. f is an arbitrary activation function (sec. 3.2).

Remark 3.2. *The function class $\sum^I(f)$ can also be written in matrix form*

$$NN(\mathbf{x}) = \mathbf{v} f(W\mathbf{x} - \theta)$$

where $\mathbf{x} \in \mathbb{R}^I$, $\mathbf{v}, \theta \in \mathbb{R}^{\bar{J}}$, and $W \in \mathbb{R}^{\bar{J} \times I}$.

Recall, that in this context the computation of the function $f(\cdot) : \mathbb{R}^{\bar{J}} \to \mathbb{R}^{\bar{J}}$ is defined component-wise (eq. 3.2).

Definition 3.3. *A function f is called a sigmoid function, if f is monotonically increasing and bounded, i.e.,*

$$f(a) \in [\alpha, \beta], \quad \textit{whereas} \quad \lim_{a \to -\infty} f(a) = \alpha \\ \textit{and} \quad \lim_{a \to \infty} f(a) = \beta$$

with $\alpha, \beta \in \mathbb{R}$ and $\alpha < \beta$.

Definition 3.4. *Let $\mathcal{C}^I$ and $\mathcal{M}^I$ be the sets of all continuous and respectively all Borel-measurable functions from $\mathbb{R}^I$ to $\mathbb{R}$. Further denote $\mathbb{B}^I$ the Borel-σ-algebra of $\mathbb{R}^I$ and $(\mathbb{R}^I, \mathbb{B}^I)$ the I-dimensional Borel-measurable space.*

$\mathcal{M}^I$ contains all functions relevant for applications. $\mathcal{C}^I$ is a subset of it. Consequently, for every Borel-measurable function f the class $\sum^I(f)$ belongs to the set $\mathcal{M}^I$ and for every continuous f to its subset $\mathcal{C}^I$.

Definition 3.5. *A subset S of a metric space (X, ρ) is ρ-dense in a subset T, if there exists, for any $\varepsilon > 0$ and any $t \in T$, $s \in S$, such that $\rho(s, t) < \varepsilon$.*

This means that every element of S can approximate any element of T with an arbitrary accuracy. In the following T and X are replaced by $\mathcal{C}^I$ and $\mathcal{M}^I$ and S by $\sum^I(f)$ with an arbitrary but fixed f. The metric ρ is chosen accordingly.

Definition 3.6. *A subset S of $\mathcal{C}^I$ is uniformly dense on a compact domain in $\mathcal{C}^I$, if, for any compact subset $K \subset \mathbb{R}^I$, S is ρ_K-dense in $\mathcal{C}^I$, where for $f, g \in \mathcal{C}^I$ $\rho_K(f, g) \equiv \sup_{x \in K} |f(x) - g(x)|$.*

Definition 3.7. *Given a probability measure μ on $(\mathbb{R}^I, \mathbb{B}^I)$, the metric $\rho_\mu : \mathcal{M}^I \times \mathcal{M}^I \to \mathbb{R}^+$ be defined as follows*

$$\rho_\mu(f,g) = \inf\{\varepsilon > 0 : \mu\{x : |f(x) - g(x)| > \varepsilon\} < \varepsilon\}.$$

Theorem 3.1. ***(Universal approximation theorem for FFNN)***
For any sigmoid activation function f, any dimension I and any probability measure μ on $(\mathbb{R}^I, \mathbb{B}^I)$, $\sum^I(f)$ is uniformly dense on a compact domain in $\mathcal{C}^I$ and ρ_μ-dense in $\mathcal{M}^I$.

Proof. The full proof of the theorem can be found in [38]. Its main steps are as follows: First the theorem is proven for an extended class of functions $\sum \Pi^I(f)$, which is defined as $\{F : \mathbb{R}^I \to \mathbb{R} : F(\mathbf{x}) = \sum_{j=1}^{\bar{J}} v_j \Pi_{k=1}^{l} f(A_j(\mathbf{x})) \quad \mathbf{x} \in \mathbb{R}^I, v_j \in \mathbb{R}, A_j \in \mathrm{A}^I, \bar{J}, l \in \mathbb{N}\}$, by applying the Stone-Weierstrass theorem [86, 94]. The transfer from the extended function class, $\sum \Pi^I(f)$, to the original one, $\sum^I(f)$, is then done with the help of the trigonometric equation. According to this, functions of the class $\sum \Pi^I(\cos)$ can be rewritten in the form of $\sum^I(\cos)$. Finally, by using the cosine squasher of Gallant and White[20] it is shown that any function of the form $\sum^I(\cos)$ can be approximated by one of $\sum^I(f)$, which proves the theorem.

The theorem states that a three-layered feedforward neural network is able to approximate any continuous function uniformly on a compact domain and any measurable function in the ρ_μ-metric with an arbitrary accuracy. The proposition is independent of the applied sigmoid activation function f (def. 3.3), the dimension of the input space I, and the underlying probability measure μ. Consequently three-layered FFNN are universal approximators.

Theorem 3.1 is only valid for FFNN with I input-, $\bar{J}$ hidden- and a single output-neuron. Accordingly, only functions from $\mathbb{R}^I$ to $\mathbb{R}$ can be approximated. However with a simple extension it can be shown that the theorem holds for networks with a multiple output (cor. 3.1).

For this, the set of all continuous functions from $\mathbb{R}^I$ to $\mathbb{R}^N$, $I, N \in \mathbb{N}$, be denoted by $\mathcal{C}^{I,N}$ and the one of (Borel-)measurable functions from $\mathbb{R}^I$ to $\mathbb{R}^N$ by $\mathcal{M}^{I,N}$ respectively. The function class $\sum^I$ gets extended to $\sum^{I,N}$ by (re-)defining the weights v_j ($j = 1, \ldots, \bar{J}$) in definition 3.2 as $N \times 1$ vectors. In matrix-form the class $\sum^{I,N}$ is then given by

$$NN(\mathbf{x}) = V f(W\mathbf{x} - \theta)$$

with $\mathbf{x} \in \mathbb{R}^I, \theta \in \mathbb{R}^{\bar{J}}, W \in \mathbb{R}^{\bar{J} \times I}$, and $V \in \mathbb{R}^{N \times \bar{J}}$. The computation of the function $f(\cdot) : \mathbb{R}^{\bar{J}} \to \mathbb{R}^{\bar{J}}$ be once more defined component-wise (rem. 3.2).

In the following, function $g : \mathbb{R}^I \to \mathbb{R}^N$ has got the elements g_k, $k = 1, \ldots, N$.

Corollary 3.1. *Theorem 3.1 holds for the approximation of functions in $\mathcal{C}^{I,N}$ and $\mathcal{M}^{I,N}$ by the extended function class $\sum^{I,N}$. Thereby the metric ρ_μ is replaced by $\rho_\mu^N := \sum_{k=1}^N \rho_\mu(f_k, g_k)$.*

Proof. [38].

Consequently three-layered multi-output FFNN are universal approximators for vector-valued functions.

3.3.2 Universal Approximation by RNN

The universal approximation theorem for feedforward neural networks (theo. 3.1) proves that any (Borel-)measurable function can be approximated by a three-layered FFNN. Based on this it is now proven that RNN in state space model form (eq. 3.5) are also universal approximators and able to approximate any open dynamical system (eq. 3.3) with an arbitrary accuracy. The proof resolves a couple of major difficulties. First the results for FFNN approximating functions (sec. 3.3.1) has to be transferred to RNN mapping open dynamical systems. Here, especially the effect of recurrence has to be taken into account. One also has to pay attention at the different dimensions. Furthermore, the output equation of the open dynamical system (eq. 3.3) is in contrast to the one of the RNN (eq. 3.5) non-linear. Therefore its non-linearity has to be incorporated into the state equation of the RNN.

Definition 3.8. *For any (Borel-)measurable function $f(\cdot) : \mathbb{R}^{\bar{J}} \to \mathbb{R}^{\bar{J}}$ and $I, N, T \in \mathbb{N}$ be $RNN^{I,N}(f)$ the class of functions*

$$\begin{aligned} \bar{\mathbf{s}}_{t+1} &= f(A\bar{\mathbf{s}}_t + B\mathbf{x}_t - \theta) \\ \bar{\mathbf{y}}_t &= C\bar{\mathbf{s}}_t \quad . \end{aligned}$$

Thereby be $\mathbf{x}_t \in \mathbb{R}^I$, $\bar{\mathbf{s}}_t \in \mathbb{R}^{\bar{J}}$, and $\bar{\mathbf{y}}_t \in \mathbb{R}^N$, with $t = 1, \ldots, T$. Further be the matrices $A \in \mathbb{R}^{\bar{J}\times\bar{J}}$, $B \in \mathbb{R}^{\bar{J}\times I}$, and $C \in \mathbb{R}^{N\times\bar{J}}$ and the bias $\theta \in \mathbb{R}^{\bar{J}}$. In the following, analogue to remark 3.2, the calculation of the function f be defined component-wise, i.e.,

$$(\bar{\mathbf{s}}_{t+1})_j = f(A_j\bar{\mathbf{s}}_t + B_j\mathbf{x}_t - \theta_j),$$

where A_j and B_j $(j = 1, \ldots, \bar{J})$ denote the $j-th$ row of the matrices A and B respectively.

It is obvious that the class $RNN^{I,N}(f)$ is equivalent to the RNN in state space model form (eq. 3.5). Analogue to its description in section 3.2 as well as definition 3.2, I stands for the number of input-neurons, $\bar{J}$ for the number of hidden-neurons and N for the number of output-neurons. $\mathbf{x}_t$ denotes the external inputs,

$\bar{\mathbf{s}}_t$ the inner states and $\bar{\mathbf{y}}_t$ the outputs of the RNN ($t = 1, \ldots, T$). The matrices A, B, and C correspond to the weight-matrices between hidden- and hidden-, input- and hidden-, and hidden- and output-neurons. f is an arbitrary activation function.

Theorem 3.2. ***(Universal approximation theorem for RNN)***
Let $g : \mathbb{R}^J \times \mathbb{R}^I \to \mathbb{R}^J$ be measurable and $h : \mathbb{R}^J \to \mathbb{R}^N$ be continuous, the external inputs $\mathbf{x}_t \in \mathbb{R}^I$, the inner states $\mathbf{s}_t \in \mathbb{R}^J$, and the outputs $\mathbf{y}_t \in \mathbb{R}^N$ ($t = 1, \ldots, T$). Then, any open dynamical system of the form

$$\begin{aligned} \mathbf{s}_{t+1} &= g(\mathbf{s}_t, \mathbf{x}_t) \\ \mathbf{y}_t &= h(\mathbf{s}_t) \end{aligned}$$

can be approximated by an element of the function class $RNN^{I,N}(f)$ (def. 3.8) with an arbitrary accuracy, where f is a continuous sigmoidal activation function (def. 3.3).

Proof. The proof is given in two steps. Thereby the equations of the dynamical system are traced back to the representation by a three-layered FFNN.

In the first step, it is concluded that the state space equation of the open dynamical system, $\mathbf{s}_{t+1} = g(\mathbf{s}_t, \mathbf{x}_t)$, can be approximated by a neural network of the form $\bar{\mathbf{s}}_{t+1} = f(A\bar{\mathbf{s}}_t + B\mathbf{x}_t - \theta)$ for all $t = 1, \ldots, T$.

Let now be $\varepsilon > 0$ and $f : \mathbb{R}^{\bar{J}} \to \mathbb{R}^{\bar{J}}$ be a continuous sigmoid activation function. Further let $K \subset \mathbb{R}^J \times \mathbb{R}^I$ be a compact set, which contains $(\mathbf{s}_t, \mathbf{x}_t)$ and $(\bar{\mathbf{s}}_t, \mathbf{x}_t)$ for all $t = 1, \ldots, T$. From the universal approximation theorem for FFNN (theo. 3.1) and the subsequent corollary (cor. 3.1) it is known that for any measurable function $g(\mathbf{s}_t, \mathbf{x}_t) : \mathbb{R}^J \times \mathbb{R}^I \to \mathbb{R}^J$ and for an arbitrary $\delta > 0$, a function

$$NN(\mathbf{s}_t, \mathbf{x}_t) = V f(W\mathbf{s}_t + B\mathbf{x}_t - \bar{\theta}),$$

with weight matrices $V \in \mathbb{R}^{J \times \bar{J}}$, $W \in \mathbb{R}^{\bar{J} \times J}$ and $B \in \mathbb{R}^{\bar{J} \times I}$ and a bias $\bar{\theta} \in \mathbb{R}^{\bar{J}}$ exists, such that

$$\sup_{\mathbf{s}_t, \mathbf{x}_t \in K} \|g(\mathbf{s}_t, \mathbf{x}_t) - NN(\mathbf{s}_t, \mathbf{x}_t)\|_\infty < \delta \quad \forall t = 1, \ldots, T. \tag{3.11}$$

As f is continuous and T finite, there exists a $\delta > 0$, such that according to the ε-δ-criterion one gets out of equation (3.11) that for the dynamics

$$\bar{\mathbf{s}}_{t+1} = V f(W\bar{\mathbf{s}}_t + B\mathbf{x}_t - \bar{\theta})$$

the following condition holds

$$\|\mathbf{s}_t - \bar{\mathbf{s}}_t\|_\infty < \varepsilon \quad \forall \ t = 1, \ldots, T. \tag{3.12}$$

Further let

$$\mathbf{s}'_{t+1} := f(W\bar{\mathbf{s}}_t + B\mathbf{x}_t - \bar{\theta})$$

which gives that

$$\bar{\mathbf{s}}_t = V\mathbf{s}'_t. \tag{3.13}$$

With the help of a variable transformation from $\bar{\mathbf{s}}_t$ to $\mathbf{s}'_t$ and the replacement $A := WV (\in \mathbb{R}^{\bar{J}\times\bar{J}})$, one gets the desired function on state $\mathbf{s}'_t$:

$$\mathbf{s}'_{t+1} = f(A\mathbf{s}'_t + B\mathbf{x}_t - \bar{\theta}) \tag{3.14}$$

Remark 3.3. *The transformation from $\bar{\mathbf{s}}_t$ to $\mathbf{s}'_t$ might involve an enlargement of the internal state space dimension.*

In the second step it is shown that the output equation $\mathbf{y}_t = h(\mathbf{s}_t)$ can be approximated by a neural network of the form $\bar{\mathbf{y}}_t = C\bar{\mathbf{s}}_t$. Thereby one has to cope with the additional challenge, to approach the non-linear function $h(\mathbf{s}_t)$ of the open dynamical system by a linear equation $C\bar{\mathbf{s}}_t$.

Let $\tilde{\varepsilon} > 0$. As h is continuous per definition, there exists an $\varepsilon > 0$, such that (according to the ε-δ-criterion) out of $\|\mathbf{s}_t - \bar{\mathbf{s}}_t\|_\infty < \varepsilon$ (eq. 3.12) follows that $\|h(\mathbf{s}_t) - h(\bar{\mathbf{s}}_t)\|_\infty < \tilde{\varepsilon}$. Consequently it is sufficient to show that $\hat{\mathbf{y}}_t = h(\bar{\mathbf{s}}_t)$ can be approximated by a function of the form $\bar{\mathbf{y}}_t = C\bar{\mathbf{s}}_t$ with an arbitrary accuracy. The proposition then follows out of the triangle inequality.

Once more the universal approximation theorem for FFNN (theo. 3.1) and the subsequent corollary (cor. 3.1) are used, which gives that equation

$$\hat{\mathbf{y}}_t = h(\bar{\mathbf{s}}_t)$$

can be approximated by a feedforward neural network of the form

$$\bar{\mathbf{y}}_t = Nf(M\bar{\mathbf{s}}_t - \hat{\theta})$$

where $N \in \mathbb{R}^{N\times\hat{J}}$ and $M \in \mathbb{R}^{\hat{J}\times J}$ be suitable weight matrices, $f : \mathbb{R}^{\hat{J}} \to \mathbb{R}^{\hat{J}}$ a sigmoid activation function, and $\hat{\theta} \in \mathbb{R}^{\hat{J}}$ a bias. According to equations (3.13) and (3.14) it is known that $\bar{\mathbf{s}}_t = V\mathbf{s}'_t$ and $\mathbf{s}'_t = f(A\mathbf{s}'_{t-1} + B\mathbf{x}_{t-1} - \bar{\theta})$. By insertion one gets

$$\begin{aligned} \bar{\mathbf{y}}_t &= Nf(M\bar{\mathbf{s}}_t - \hat{\theta}) \\ &= Nf(MV\mathbf{s}'_t - \hat{\theta}) \\ &= Nf(MVf(A\mathbf{s}'_{t-1} + B\mathbf{x}_{t-1} - \bar{\theta}) - \hat{\theta}). \end{aligned} \tag{3.15}$$

Using again theorem 3.1 equation (3.15) can be approximated by

$$\tilde{\mathbf{y}}_t = Df(E\mathbf{s}'_{t-1} + F\mathbf{x}_{t-1} - \tilde{\theta}) \quad , \tag{3.16}$$

with suitable weight matrices $D \in \mathbb{R}^{N \times \bar{\bar{J}}}$, $E \in \mathbb{R}^{\bar{\bar{J}} \times \bar{J}}$, and $F \in \mathbb{R}^{\bar{\bar{J}} \times I}$, a bias $\tilde{\theta} \in \mathbb{R}^{\bar{\bar{J}}}$, and a (continuous) sigmoid activation function $f : \mathbb{R}^{\bar{\bar{J}}} \to \mathbb{R}^{\bar{\bar{J}}}$.

If one further sets

$$\mathbf{r}_{t+1} := f(E\mathbf{s}'_t + F\mathbf{x}_t - \tilde{\theta}) \quad (\in \mathbb{R}^{\bar{\bar{J}}})$$

and enlarges the system equations (3.14) and (3.16) about this additional component, one achieves the following form

$$\begin{aligned} \begin{pmatrix} \mathbf{s}'_{t+1} \\ \mathbf{r}_{t+1} \end{pmatrix} &= f\left(\begin{pmatrix} A & 0 \\ E & 0 \end{pmatrix} \begin{pmatrix} \mathbf{s}'_t \\ \mathbf{r}_t \end{pmatrix} + \begin{pmatrix} B \\ F \end{pmatrix} \mathbf{x}_t - \begin{pmatrix} \bar{\theta} \\ \tilde{\theta} \end{pmatrix} \right) \\ \tilde{\mathbf{y}}_t &= (0 \quad D) \begin{pmatrix} \mathbf{s}'_t \\ \mathbf{r}_t \end{pmatrix}. \end{aligned}$$

Their equivalence to the original equations (3.14) and (3.16) is easy to see by a component-wise computation.

Finally out of

$$\begin{aligned} &\tilde{J} := \bar{J} + \bar{\bar{J}}, \tilde{\mathbf{s}}_t := \begin{pmatrix} \mathbf{s}'_t \\ \mathbf{r}_t \end{pmatrix} \in \mathbb{R}^{\tilde{J}}, \\ &\tilde{A} := \begin{pmatrix} A & 0 \\ E & 0 \end{pmatrix} \in \mathbb{R}^{\tilde{J} \times \tilde{J}}, \tilde{B} := \begin{pmatrix} B \\ F \end{pmatrix} \in \mathbb{R}^{\tilde{J} \times I}, \\ &\tilde{C} := (0 \quad D) \in \mathbb{R}^{N \times \tilde{J}} \text{ and } \theta := \begin{pmatrix} \bar{\theta} \\ \tilde{\theta} \end{pmatrix} \in \mathbb{R}^{\tilde{J}}, \end{aligned}$$

follows

$$\begin{aligned} \tilde{\mathbf{s}}_{t+1} &= f(\tilde{A}\tilde{\mathbf{s}}_t + \tilde{B}\mathbf{x}_t - \theta) \\ \tilde{\mathbf{y}}_t &= \tilde{C}\tilde{\mathbf{s}}_t \quad . \end{aligned} \tag{3.17}$$

Equation (3.17) is apparently an element of the function class $RNN^{I,N}(f)$. Thus, the theorem is proven.

□

A further extension of the proof to other open dynamical systems and normalised RNN [106] can be found in [79].

3.4 Training of Recurrent Neural Networks

Training of neural networks is an important factor, as it is essential for the quality, speed and robustness in approximation of the networks. Especially with RNN many researchers apparently see a challenge in finding an optimal training algorithm and even claim that certain structures are impossible to learn with those

networks [36]. For that reason also new recurrent networks have been developed, like the mentioned echo-state [39] and LSTM [37] networks, which aim at circumventing the apparent learning problem. Besides many different algorithms for recurrent but especially for feedforward neural networks have been proposed, each with a different focus on performance improvement, e.g. [57, 58, 67, 69, 83].

To show that training of RNN is not a major problem or particularly difficult, its main aspects are presented in this section. First the already mentioned shared weights extension of the standard backpropagation algorithm [71, 95, 97] is briefly explained (sec. 3.4.1). Subsequent, in section 3.4.2, two different learning methods based on the gradient calculation of the backpropagation algorithm are given. Here, the focus is on accuracy in combination with robustness instead of speed, as especially for system identification and reinforcement learning, the quality of the performance is considered as the crucial issue. In section 3.4.3 finally the often doubted and criticised aspect of long-term learning is analysed. It is shown that RNN are in contrast to an often cited statement well able to identify and learn long-term dependencies.

3.4.1 Shared Weight Extended Backpropagation

Originally the backpropagation algorithm was invented by Paul Werbos in his PhD-thesis [95]. Still, it was not widely recognised until the (re-)invention of Rumelhart and McClelland in 1986 [71]. Further developments about the algorithm are summarised in [97].

The backpropagation algorithm is an efficient method for the gradient calculation of the error function with respect to the weights of a neural network. The error gradient can then be used to optimise the network's weights with the objective to minimise the difference between output and target. Hereby nearly any kind of mathematical optimisation method [24, 48] can be applied, whereas a couple, like pattern-by-pattern (sec. 3.4.2.1) and vario-eta learning (sec. 3.4.2.2), have proven their practicality.

For an application of the standard backpropagation algorithm to the presented RNN only a slight modification is necessary. Due to unfolding in time and shared weights (sec. 3.2) the principle structure of the algorithm remains unchanged [71].

The required adjustments in comparison to standard backpropagation are in the following illustrated on the example of three-layered FFNN. Thereby, the assumption is made that the weight matrices between input and hidden layer $W \in \mathbb{R}^{\bar{J} \times I}$ with the entries w_{ji} and between hidden and output layer $V \in \mathbb{R}^{N \times \bar{J}}$ with the entries v_{nj} are shared. For two arbitrary but fixed weights w_{ji} and v_{nj}, this implies that

$$w := w_{ji} = v_{nj}$$

Assuming the quadratic error function[3]

$$E(w) = \frac{1}{2}\|\bar{\mathbf{y}} - \mathbf{y}\|_2^2 = \frac{1}{2}\sum_{n=1}^{N}(\bar{y}_n - y_n)^2 \tag{3.18}$$

the global partial derivative to the weight w is derived by making use of the product- and chain-rule:

$$\begin{aligned}\frac{\partial E}{\partial w} &= (\bar{y}_n - y_n)\frac{\partial \bar{y}_n}{\partial \bar{y}_n^{in}}\frac{\partial \bar{y}_n^{in}}{\partial v_{nj}} \\ &+ \sum_{n=1}^{N}(\bar{y}_n - y_n)\frac{\partial \bar{y}_n}{\partial \bar{y}_n^{in}}\frac{\partial \bar{y}_n^{in}}{\partial \bar{s}_j^{out}}\frac{\partial \bar{s}_j^{out}}{\partial \bar{s}_j^{in}}\frac{\partial \bar{s}_j^{in}}{\partial w_{ji}} \\ &= (\bar{y}_n - y_n)\bar{s}_j^{out} + f'(\bar{s}_j^{in})\sum_{n=1}^{N} v_{nj}(\bar{y}_n - y_n)x_i^{out}\end{aligned} \tag{3.19}$$

Taking a deeper look at equation 3.19 it turns out that the calculation of the global partial derivative of weight w is simply the sum of the two local ones of the weights v_{nj} and w_{ji} [28, 71]. Those can be easily determined by the standard backpropagation algorithm [95, 97]. Therefore the extension of the backpropagation algorithm to shared weights is simply done by adding a summation of the respective shared local partial derivatives.

The notation of equation 3.19 can be simplified by setting

$$\begin{aligned}\Delta_n &= (\bar{y}_n - y_n) \\ \delta_j &= f'(\bar{s}_j^{in})\sum_{n=1}^{N}\Delta_n v_{nj} \quad .\end{aligned}$$

Using those, equation 3.19 becomes by insertion:

$$\frac{\partial E}{\partial w} = \Delta_n \cdot \bar{s}_j^{out} + \delta_j \cdot x_i^{out}$$

Figure 3.6 depicts the extended gradient calculation. Analogue to the standard backpropagation algorithm [95, 97], the local partial derivatives of the weights v_{nj} and w_{ji} are determined by the forward and backward flow of the network. However, the local derivatives are saved separately for each weight during the backward iteration. Not until all those are calculated, the extended algorithm determines the complete partial derivative of a weight w by summing the respective stored local ones [28, 71]. Proceeding this way also the backpropagation algorithm for shared weights stays local, i.e., it only applies values calculated at the

[3]Here, for simplicity the pattern index t is omitted.

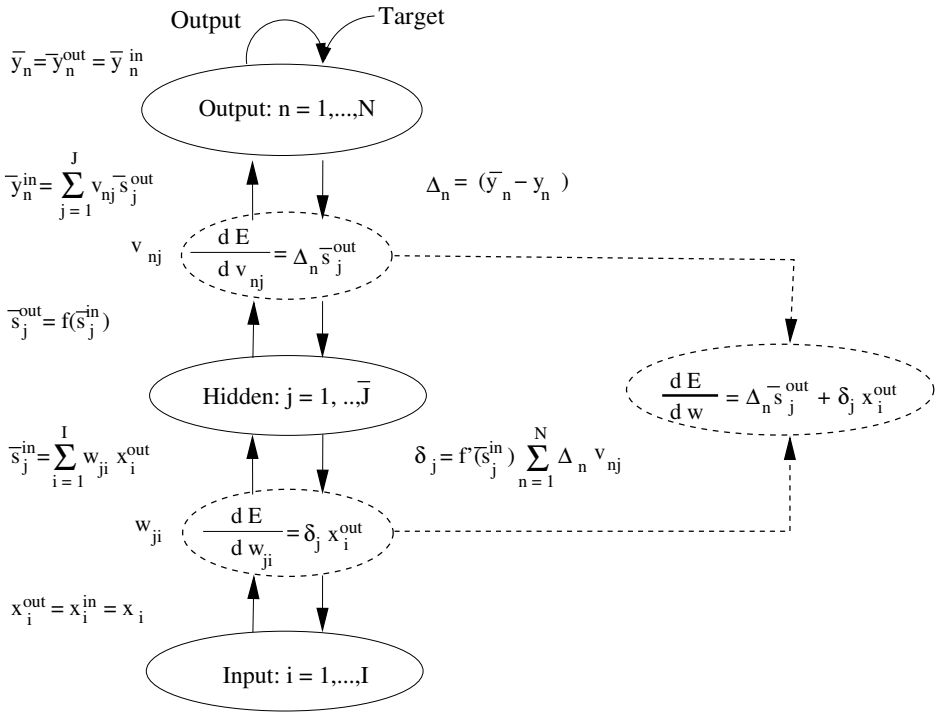


Figure 3.6: Extension of the backpropagation algorithm to shared weights on the example of a simple three-layered FFNN with the constraint that $w := v_{nj} = w_{ji}$. For simplicity the pattern index t is omitted [28].

required position and not somewhere else in the architecture. This is important for the extendability of the algorithm and respectively the recurrent networks.

The application of the extended backpropagation algorithm to RNN is due to unfolding in time and the described connection to FFNN straight forward [28].

3.4.2 Learning Methods

In the following two different learning methods for neural networks are shortly presented: pattern-by-pattern (sec. 3.4.2.1) and vario-eta learning (sec. 3.4.2.2). For simplification, the learning rate is assumed to be fixed. In practice it is also often either reduced manually during training or adjusted according to an algorithm like simulated annealing [43]. As already pointed out, several other learning algorithms for neural networks have been developed, like Rprop [67, 69] or rapid stochastic gradient descent [83]. For recurrent neural networks some of them are summarised in Pearlmutter [58]. However, the following two have shown to be very useful for training the regarded RNN (sec. 3.2). They have the particular advantage that they implicitly posses a stochastic term, which allows to leave local

minima for a global one. Furthermore, they both regularise learning through an implicit penalty term, which avoids large curvature of the error curve and consequently favours flat minima. This assures a good performance also under noisy data, because it prevents that a small weight shift might result in a large increase of the error value [57]. Besides, they are easily scalable into high dimensions and only require local gradient information, which is both important for the extendability of the networks.

3.4.2.1 Pattern-by-Pattern Learning

The pattern-by-pattern learning algorithm corresponds to (standard) gradient descent [24, 48] with the important difference that the weights are updated after each training pattern [57]. Thus, the batch size is one, which results in the following learning rule for an arbitrary but fixed weight $w_l \in \mathbb{R}$ ($l = 1, \ldots, L$, where $L \in \mathbb{N}$ denotes the number of weights in the network), and for each training pattern $t = 1, \ldots, T$

$$w_l \leftarrow w_l - \eta \frac{\partial E_t}{\partial w_l} = w_l - \eta g_{l_t}, \tag{3.20}$$

where $\eta \in \mathbb{R}^+$ denotes the learning rate and $g_{l_t} = \frac{\partial E_t}{\partial w_l}$ the gradient for a pattern t with respect to w_l (sec. 3.4.1). Thereby, the pattern are generally chosen randomly out of the training data to explore the solution space more efficiently [57].

Rewriting equation 3.20, with $g_l = \frac{\partial E}{\partial w_l} = \frac{1}{T} \sum_{t=1}^{T} g_{l_t}$ as the cumulative gradient, it becomes obvious that the pattern-by-pattern learning rule exhibits a stochastic component [57]:

$$w_l \leftarrow w_l - \eta g_l - \eta (g_{l_t} - g_l)$$

In this form the learning rule consists of the cumulative gradient g_l, with a tendency to local minima, and a perturbation term $(g_{l_t} - g_l)$, which acts against it. The latter is in contrast to g_l unequal to zero in a local minima. Therefore it helps to leave a local minimum for a global one during the learning process.

For the mentioned regularisation effect one regards the expected value of the error function $E(w_l)$ (eq. 3.18) while learning. Around local minima one can assume that the gradients g_{l_t} are i.i.d. (independent and identically distributed) with mean zero and a variance vector $\text{var}(g_l)$. Further assuming that the first and second derivatives are stable close to local minima, the expected value of the error function $[E(w_l)]$ can, by Taylor expansion, be approximated as follows

$$[E(w_l)] \approx \frac{1}{T} \sum_{t=1}^{T} E(w_l - \eta g_{l_t}) = E(w_l) + \frac{\eta^2}{2} \sum_{l=1}^{L} \text{var}(g_l) \frac{\partial^2 E}{\partial w_l^2} \quad . \tag{3.21}$$

Consequently pattern-by-pattern learning possesses an implicit local penalty term $\text{var}(g_l)$, which represents the stability or respectively uncertainty in the

weights w_l ($l = 1, \ldots, L$). In a local minimum the sum of the gradients of a particular weight w_l is zero, whereas the variance var(g_l) can be very large. In this case the solution is susceptible to disturbances. The regularising penalty term now causes a tendency towards flat or stable minima and therefore decreases the uncertainty through varying gradient information [57].

3.4.2.2 Vario-Eta Learning

The vario-eta learning rule equals to a stochastic approximation of the Newton method [24, 48] and therefore belongs to the class of quasi-Newton methods [24, 48]. Thereby the inverse of the Hessian is approximated or respectively replaced, by the standard deviation of the particular gradients g_{l_t} for each weight w_l ($l = 1, \ldots, L$). In doing so, vario-eta uses, according to its name, a variable factor, $\frac{1}{\sqrt{\frac{1}{T}\sum_{t=1}^{T}(g_{l_t}-g_l)^2}}$ in addition to the learning rate η [57]. Consequently an arbitrary but fixed weight w_l is updated as follows

$$w_l \leftarrow w_l - \frac{\eta}{\sqrt{\frac{1}{T}\sum_{t=1}^{T}(g_{l_t}-g_l)^2}} \sum_{t=1}^{N} g_{l_t}$$

where $N \leq T$ denotes the implemented batch size. The additional variable factor effects that the learning rate is scaled according to the standard deviation of the several gradients g_{l_t}. It is increased when the standard deviation is low and decreased when it is high. Therefore the achievement of an optimum is accelerated [28]. The batch size is generally kept small to achieve, analogue to pattern-by-pattern learning (sec. 3.4.2.1), a stochastic effect during learning [28, 57].

Note that the effected re-scaling of the gradient in every time step also contradicts the alleged vanishing of the gradient flow in RNN [8, 35, 36]. Here, the scaling factor serves as a reprocessing of the error information for each weight, independent of its position in the network [57, 109]. A more detailed analysis of the corresponding learning of long-term dependencies is given in section 3.4.3.

Similar to pattern-by-pattern learning (sec. 3.4.2.1) also vario-eta possess a regularisation towards stable minima. Analogue to equation 3.21, with batch size $N = 1$ the weight update of this method, $-\frac{\eta}{\sqrt{\frac{1}{T}\sum_{t=1}^{T}(g_{l_t}-g_l)^2}} g_{l_t}$, leads to an expected value of the error function

$$[E(w_l)] \approx \frac{1}{T}\sum_{t=1}^{T} E(w_l - \frac{\eta}{\sqrt{\frac{1}{T}\sum_{t=1}^{T}(g_{l_t}-g_l)^2}} g_{l_t}) = E(w_l) + \frac{\eta^2}{2}\sum_{l=1}^{L}\frac{\partial^2 E}{\partial w_l^2} \quad .$$

However, in comparison to pattern-by-pattern learning this penalty term is only global, as the (local) variance term var(g_l) has been cancelled down. This can be

of disadvantage especially in high dimensions [57].

Due to the different properties of vario-eta and pattern-by-pattern learning, in Neuneier and Zimmermann [57] an iterative application is proposed. Here, the network is first trained to a minimum with vario-eta and afterwards optimised by pattern-by-pattern learning.

3.4.3 Learning Long-Term Dependencies

Despite the presented properties and advantages of RNN unfolded in time, there is often a negative attitude towards them. One reason is, that it has been claimed by several authors that RNN are unable to identify and learn long-term dependencies of more than ten time steps [8, 35, 36]. To overcome the stated dilemma new forms of recurrent networks like for example the already mentioned LSTM networks [37] were developed. Still, these networks do not offer the described correspondence, i.e., the mutual transferability, between equations and architectures as RNN unfolded in time do.

However, the analysis in the mentioned papers [8, 35, 36] were all based on basic RNN architectures simulating closed dynamical systems, which do not consider any external inputs. Even more important, they were made from a static perspective, which means that for the presented calculations only RNN with fixed weights were assumed whereas the effect of learning and weight adaption was not taken into account. In the following the statement that RNN unfolded in time and trained with a shared weight extension of the backpropagation algorithm (sec. 3.4.1) are in general unable to learn long-term dependencies is therefore refuted. It is shown that basic RNN (sec. 3.2) have no difficulties with an identification and learning of past-time information within the data, which is more than ten time steps apart. In addition it is pointed out that by using shared weights, training of these networks even helps to overcome the problem of a vanishing gradient [8, 35, 36] as the networks possess a self-regularisation characteristic, which adapts the internal error backflow.

A very simple but well-known problem is used to demonstrate the ability of learning long-term dependencies of RNN (sec. 3.2): the prediction of periodic indicators in a time series. Therefore time series of 10000 random values, which are uniformly distributed on an interval $[-r, r]$ with $r \in \mathbb{R}$ and $0 \leq r < 1$, were created. Every d-th value, with $d \in \mathbb{N}$ is 1. By construction these time indicators are the only predictable values for the network. Consequently, for a successful solution to the problem the network has to remember the occurrence of the last 1, d-time steps before in the time series. In other words, it has to be able to learn long-term dependencies. The higher the d the longer memory is necessary. The first 5000 data points were used for training whereas the other half served for testing.

Similar problems have already been studied in [35] and [37]. In both papers the performance of the thereby considered recurrent networks trained with backpropagation through time [102] has been tested to be unsatisfactory and the authors concluded that RNN are not suited for the learning of long-term dependencies.

3.4.3.1 Model Description

An RNN (sec. 3.2) with one input $\mathbf{x}_\tau$ per time step $\tau \leq t$ in the past and a single output $\mathbf{y}_{t+1}$ in the future was applied. In contrast to the descriptions in section 3.2 no outputs were implemented in the past part of the network, as those would not help to solve the problem. This implies that the gradient information of the error function had to be propagated back from the future output to all past time steps. It also avoided a superposition of the long-term gradient information with a local error flow in the past. Therefore the omission of outputs in the past also eased the analysis of the error backflow.

The network was unfolded a hundred time steps into the past. No overshooting was implemented. This kept the RNN as simple as possible to show that even such a basic RNN is able to learn long-term dependencies. Thus, total unfolding amounted to 101 time steps. The dimension of the internal state $\bar{\mathbf{s}}$ was set to 100, which was equivalent to the amount of past unfolding. Input and output were one dimensional consisting of the time series information at a time. The architecture of the network is depicted in figure 3.7.

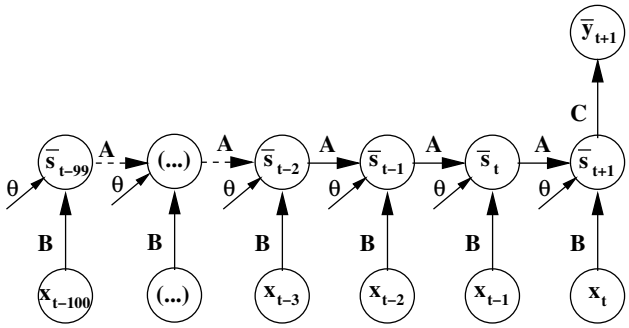


Figure 3.7: RNN architecture used for the long-term learning experiment. According to the problem setting it is $A \in \mathbb{R}^{100\times100}$, $B \in \mathbb{R}^{100\times1}$, $C \in \mathbb{R}^{1\times100}$, and $\theta \in \mathbb{R}^{100}$.

The weights were initialised randomly with a uniform distribution on $[-0.2, 0.2]$. In all hidden units the hyperbolic tangent was implemented as activation function f. Furthermore, the quadratic error function was used to minimise the difference between network output and target (eq. 3.6). The RNN was trained

with the shared weight extension of the backpropagation algorithm (sec. 3.4.1) in combination with pattern-by-pattern learning (sec. 3.4.2.1). The learning rate η was set to 10^{-4}, which is a good trade-off between speed and accuracy. The learning was restricted to this rather simple algorithm to strengthen the significance of the experiments. Otherwise also vario-eta learning (sec. 3.4.2.2) could be applied, which, as already mentioned, inherently avoids a vanishing gradient.

3.4.3.2 Results

An error limit was defined, which marks the optimal achievable error for each problem plus a 10% tolerance. For $r > 0$ it is calculated by the variance of the uniform distribution given a certain noise range r, assuming no error for the time indicators in every d-th time step and adding 10%. For $r = 0$ it is set to 0.0001, which gives together:

$$\text{error limit} = \begin{cases} 0.0001 & \text{for} \quad r = 0 \\ 1.1 \cdot \frac{d-1}{d} \cdot \frac{r^2}{3} & \text{for} \quad r > 0 \end{cases} \tag{3.22}$$

Table 3.1 summarises the results for different time gaps d and several noise ranges r. It shows the median, the average number (mean) and the standard deviation (STD) of epochs the RNN needed to pass the error limit (eq. 3.22) on the test set for a minimum of 15 trials each. Hereby, not the actual value but rather the fact that the networks are able to learn the given task within a limited number of epochs is of importance. As already pointed out, the former could most likely be decreased by applying a problem-dependent learning method.

The results demonstrate the capability of a basic RNN to learn long-term dependencies of $d = 40$, 60 and even 100, which is obviously more than the often cited limit of ten time steps [36]. As expected, a larger gap d resulted in more learning epochs for the RNN to succeed. Also a higher noise range, i.e., a larger uniform distribution of the data, made it more challenging for the network to identify the time indicators. Still, even in more difficult settings, the RNN captured the structure of the problem very quickly.

Using smaller dimensions for the internal state $\bar{\mathrm{s}}$ and hence for the transition matrix A increased the number of epochs necessary to learn the problem. This is due to the fact that the network needs a certain dimension to store long-term information. So e.g., with a hundred dimensional matrix the network can easily store a time gap of $d = 100$ in form of a simple shift register. Downsizing the dimension forces the network to build up more complicated internal matrix structures, which take more learning epochs to develop [78].

time gap d	range r	# epochs RNN Median	Mean	STD
40	0.0	27	29	19
40	0.1	41	70	103
40	0.2	27	73	114
40	0.4	93	158	168
60	0.0	131	280	348
60	0.1	132	317	362
60	0.2	298	472	482
60	0.4	596	584	284
100	0.0	50	106	231
100	0.1	30	181	319
100	0.2	23	193	318
100	0.4	123	225	273

Table 3.1: Median, average number (Mean) and standard deviation (STD) of learning epochs the RNN needed to pass the error limit (eq. 3.22) for a minimum of 15 trials each, i.e. to solve the problem, on the test set for different time gaps d and noise ranges r.

3.4.3.3 Analysis of the Backpropagated Error

To put the claim of a vanishing gradient in RNN unfolded in time and trained with backpropagation [36] into perspective the backpropagated error within the applied networks was analysed. It turned out that under certain conditions vanishing gradients do indeed occur, but are only a problem if one puts a static view on the networks like it has been done in [35, 36]. Studying the development of the error flow during the learning process it could be observed that the RNN itself has a regularising effect, i.e., it is able to prolong its information flow and consequently to solve the problem of a vanishing gradient. The concept of shared weights (sec. 3.2) is mainly responsible for this self-regularisation behaviour, as it constrains the network to change weights (concurrently) in every unfolded time step according to several different error flows. This allows the RNN to adapt the gradient information flow.

Similar to the analysis in [35] and [36] it could further be confirmed that the occurrence of a vanishing gradient is dependent on the values of the weights within the weight matrices. By initialising with different weight values it turned out that an initialisation with a uniform distribution in $[-0.2, 0.2]$ is a good choice for the tested networks. In these cases no vanishing gradient could be experienced. In contrast, when initialising the RNN only within $[-0.1, 0.1]$, the gradient vanished in the beginning of the learning process. Nevertheless, during the learning

process the RNN itself solved this problem by changing the weight values. Figure 3.8 shows an exemplary change of the gradient information flow during the learning process.

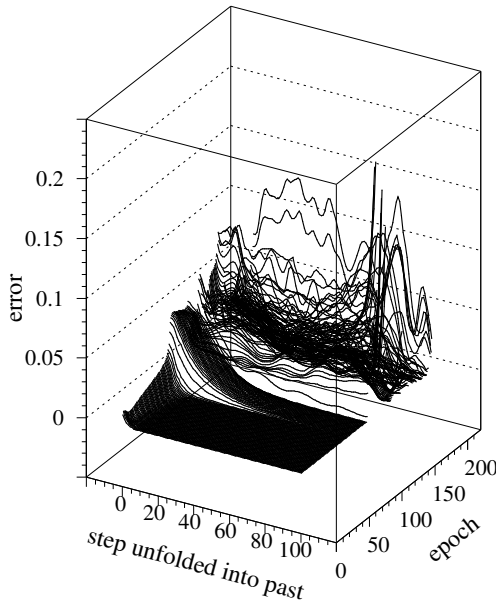


Figure 3.8: Exemplary adaptation of the gradient error flow during the learning process of an RNN, which has been initialised with small weights, i.e., within $[-0.1, 0.1]$: The graph shows that for about the first 100 learning epochs the gradient vanishes very quickly. After that the error information distributes more and more over the different unfolding steps, i.e., the network prolongs its memory span. Finally after about a 150 epochs the error information is almost uniformly backpropagated to the last unfolded time step.

A further analysis of an optimal weight initialisation is given section 3.5.3.

3.5 Improved Model-Building with RNN

In practical applications different approaches in model-building with RNN have shown to be very useful, e.g. [29, 104, 107, 110]. Although it is mostly difficult to prove their evidence, they have empirically demonstrated their effectiveness.

One aspect is the dealing with uncertainty, which can disturb the development of the internal dynamics and derogate the quality of the system identification. In

particular, the input data itself might be corrupted or noisy (sec. 3.5.1). Moreover, in the framework of RNN finitely unfolded in time one is confronted with the uncertainty of the initial state (sec. 3.5.2). Both types are also relevant for the application of RNN to RL problems (chap. 4). Furthermore, the topic of weight initialisation (sec. 3.5.3) is discussed. As already pointed out, it is important for a proper training of RNN, especially with a focus on long-term learning.

3.5.1 Handling Data Noise

In most real-world applications data is often corrupted or noisy. Reasons for this are multifarious. Examples are inexact or forgotten measurements, lost data or just an unstable system development. As one of those generally applies to any application, a couple of approaches have been developed, which aim at a minimisation of this uncertainty in data quality.

One of those methods is so-called cleaning noise. It improves the model's learning behaviour by correcting corrupted or noisy input data. The method is an enhancement of the cleaning technique, which is described in detail in [57, 108]. In short, cleaning considers the inputs as corrupted and adds corrections to the inputs if necessary. However, one generally wants to keep the cleaning correction as small as possible. This leads to an extended error function

$$E_t^{\mathbf{y},\mathbf{x}} = \frac{1}{2}\|\bar{\mathbf{y}}_t - \mathbf{y}_t\|_2^2 + \|\bar{\mathbf{x}}_t - \mathbf{x}_t\|_2^2 = E_t^{\mathbf{y}} + E_t^{\mathbf{x}} \quad \rightarrow \min_{\bar{\mathbf{x}}_t, w} \quad ,$$

where in addition to equation 3.18 also a correction of the input variables $\mathbf{x}_t$ is allowed. Hereby $\bar{\mathbf{x}}_t \in \mathbb{R}^I$ stands for the corrected input and w represents all adjustable weights in the respective network.

The new error function does not change the weight adaption rule (sec. 3.4). To calculate the cleaned input only the correction vector $\rho_t \in \mathbb{R}^I$ for all input data of the training set is needed:

$$\text{Cleaning:} \quad \bar{\mathbf{x}}_t = \mathbf{x}_t + \rho_t$$

The update rule for these cleaning corrections, initialised with $\rho_t = 0$, can be derived from typical adaption sequences

$$\bar{\mathbf{x}}_t \leftarrow \bar{\mathbf{x}}_t - \eta \frac{\partial E_t^{\mathbf{y},\mathbf{x}}}{\partial \mathbf{x}_t} \quad ,$$

leading to

$$\rho_t \leftarrow (1-\eta)\rho_t - \eta \frac{\partial E_t^{\mathbf{y}}}{\partial \mathbf{x}_t} \quad .$$

This is a non-linear version of the error-in-variables concept from statistics [21].

All required information, especially the residual error $\frac{\partial E_t^{\mathbf{y},\mathbf{x}}}{\partial \mathbf{x}_t}$, is derived from training the network with backpropagation (sec. 3.4.1), which makes the computational effort negligible. In this way the corrections are performed by the model itself and not by applying external knowledge. This refers to the so-called "observer-observation-dilemma", which is an important problem in (neural network) model-building [108].

If the data is not only corrupted but also noisy, it is useful to add an extra noise vector $-\rho_\tau \in \mathbb{R}^I$ $(\tau = 1, \ldots, T)$ to the cleaned value as this allows to represent the whole input distribution to the network instead of using only one particular realisation [57]:

$$\text{Cleaning Noise:} \quad \bar{\mathbf{x}}_t = \mathbf{x}_t + \rho_t - \rho_\tau$$

The noise vector ρ_τ is hereby a randomly chosen row vector $\{\rho_{i\tau}\}_{i=1,\ldots,I}$ of the cleaning matrix

$$C_{Cl} := \begin{bmatrix} \rho_{11} & \cdots & \cdots & \cdots & \rho_{I1} \\ \rho_{12} & \ddots & & & \rho_{I2} \\ \vdots & & \rho_{it} & & \vdots \\ \vdots & & & \ddots & \vdots \\ \rho_{1T} & \cdots & \cdots & \cdots & \rho_{IT} \end{bmatrix},$$

which stores the input error corrections of all data patterns. The matrix has the same size as the pattern matrix, as the number of rows equals the number of patterns T and the number of columns equals the number of inputs I.

A variation on the cleaning noise method is so-called local cleaning noise. Cleaning noise adds to every component of the input vector the same noise term $-\rho_\tau$. In contrast, local cleaning noise is able to differentiate componentwise. Hence, it corrects each component of the input vector $\mathbf{x}_{it}$ individually by a cleaning correction and a randomly taken entry $\rho_{i\tau}$ of the corresponding column $\{\rho_{it}\}_{t=1,\ldots,T}$ of the cleaning matrix C_{Cl}:

$$\bar{\mathbf{x}}_{it} = \mathbf{x}_{it} + \rho_{it} - \rho_{i\tau}$$

The advantage of the local cleaning technique is that, with the increased number of (local) correction terms $(T \cdot I)$, one can cover higher dimensions. In contrast, with the normal cleaning technique the dimension is bounded by the number of training patterns T, which can be insufficient for high-dimensional problems, where only a limited amount of training data is available.

3.5.2 Handling the Uncertainty of the Initial State

A difficulty with finite unfolding in time RNN is to find a proper initialisation for the first state vector. An obvious solution is to set the last unfolded state

$\bar{\mathbf{s}}_{m^-} \in \mathbb{R}^{\bar{J}}$ to zero. However, this implicitly assumes that the unfolding includes enough (past) time steps such that the misspecification of the initialisation phase is compensated along the state transitions. In other words, one supposes that the network accumulates sufficient information over time, and thus can eliminate the impact of the arbitrary initial state on the network outputs.

The model can though be improved by making the unfolded RNN less sensitive to the unknown initial state $\bar{\mathbf{s}}_{m^-}$. For this purpose an initialisation is proposed for which the interpretation of the state recursion is consistent over time.

Here, a noise term is added to the last unfolded state vector $\bar{\mathbf{s}}_{m^-}$ to stiffen the model against the uncertainty of the unknown initial state. In practise a fixed noise term that is drawn from a predetermined noise distribution has shown to be inadequate, as in particular the associated variance is difficult to estimate. Therefore, according to the cleaning noise method (sec. 3.5.1), an adaptive noise term is added, which fits best the volatility of the unknown initial state. As explained in section 3.5.1, the characteristics of the adaptive noise term are automatically determined as a by-product of the error backpropagation algorithm (sec. 3.4.1).

The basic idea is as follows: The residual error $\rho_t \in \mathbb{R}^{\bar{J}}$ of an arbitrary but fixed pattern t ($t = 1, \ldots, T$) as measured at the last unfolded state vector $\bar{\mathbf{s}}_{m^-}$ can be interpreted as a measure for the uncertainty originating from missing information about the true initial state vector. If one disturbs $\bar{\mathbf{s}}_{m^-}$ with a noise term, which follows the distribution of the residual error of the network, the uncertainty about the unknown initial state during system identification can be diminished.

Technically, the noise is introduced into the model via an additional input layer. Its dimension is equal to that of the internal state. The input values are fixed to zero over time. The desensitisation of the network from the initial state vector $\bar{\mathbf{s}}_{m^-}$ can therefore be seen as a self-scaling stabiliser of the modelling. The noise vector $\rho_\tau \in \mathbb{R}^{\bar{J}}$ is drawn randomly from the observed residual errors, without any prior assumption on the underlying noise distribution.

The effect becomes easier to understand if one regards the model's internal development over time. In general, a time-discrete state trajectory forms a sequence of points over time. Such a trajectory is comparable to a thread in the internal state space. Most notably, it is very sensitive to the initial state vector $\bar{\mathbf{s}}_{m^-}$. If noise is applied to $\bar{\mathbf{s}}_{m^-}$, the space of all possible trajectories becomes a tube in the internal state space (fig. 3.9). Due to the characteristics of the adaptive noise term, which decreases over time, the tube contracts. This enforces the identification of a stable dynamical system. Consequently, the finite volume trajectories act as a regularisation and stabilisation of the dynamics.

Table 3.2 gives an overview of several initialisation techniques, which have been developed and examined. The first three methods have already been explained in section 3.5.1. The idea behind the initialisation with start noise is, that one abstains from a cleaning correction but solely focuses on the noise term. In

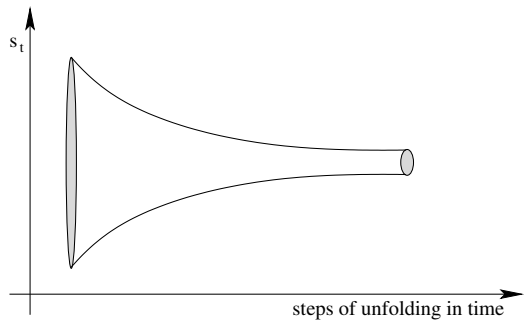


Figure 3.9: Creating a tube in the internal state space by applying noise to the initial state.

all cases local always corresponds to the individual application of a noise term to each component of the initial state $\bar{\mathbf{s}}_{m^-}$ (sec. 3.5.1). From top to bottom the methods listed in table 3.2 use fewer information about the training set. Hence, start noise is preferred when only a limited amount of training data is available.

Cleaning:	$\bar{\mathbf{s}}_{m^-} = 0 + \rho_t$
Cleaning Noise:	$\bar{\mathbf{s}}_{m^-} = 0 + \rho_t - \rho_\tau$
Local Cleaning Noise:	$\bar{\mathbf{s}}_{m_i^-} = 0 + \rho_{it} - \rho_{i\tau}$
Start Noise:	$\bar{\mathbf{s}}_{m^-} = 0 + \rho_\tau$
Local Start Noise:	$\bar{\mathbf{s}}_{m_i^-} = 0 + \rho_{i\tau}$

Table 3.2: Overview of initialisation techniques

3.5.3 Optimal Weight Initialisation

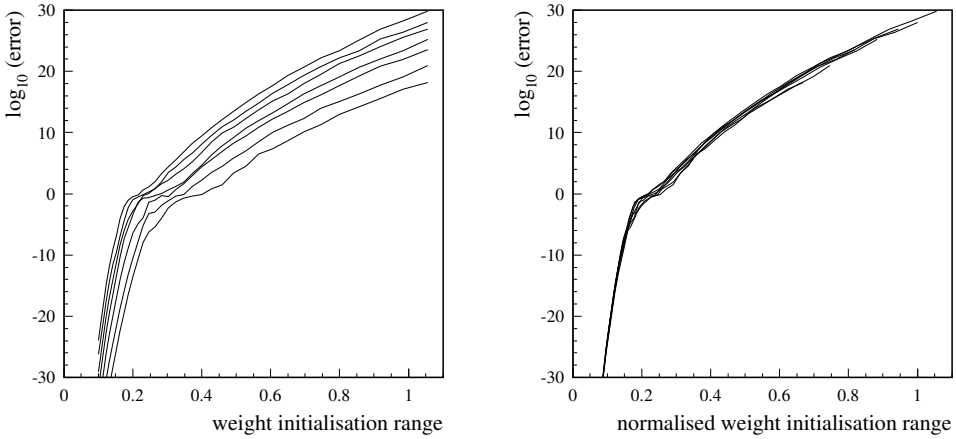
In section 3.4.3.3 it was already stated that a proper weight initialisation is of importance for the learning of the networks. It turned out that choosing the distribution interval too small can lead to a vanishing gradient in the beginning of learning. In contrast, a too large one can generate very high values in the backflow. In most cases this can be corrected during training (sec. 3.4.3.3), but it generally leads to long computational times. For extremely small or large values the network might even be unable to solve the problem.

For that reason the effects on the backpropagated error flow of different weight initialisations in RNN were analysed. This can be done by measuring the error values backpropagated to the last unfolded time step.

For the experiment the same RNN as in section 3.4.3.2 with internal state dimensions of $\dim(\mathbf{s}) = 40, 50, 60, 70, 80, 90$ and 100 was taken and the error

value backpropagated to the hundredth unfolded time step for different weight initialisations was calculated. The problem setting was as before $d = 40$ and $r = 0.1$. Figure 3.10(a) plots the measured logarithmic error values against different ranges of uniform distributed weight initialisations, each respectively averaged over ten different random initialisations. Note that for this no learning, i.e., weight adaption, was involved, which is equivalent to $\eta = 0$ (sec. 3.4). In the plot the more left the curve is the larger is its internal state dimension.

The test confirmed that for the 100-dimensional RNN as used in the long-term experiment (sec. 3.4.3.1) an initialisation with weights in $[-0.2, 0.2]$ generates a stable error backflow with neither a vanishing nor an exploding gradient. For those values a reasonable error E ($0.0001 < E < 1$) is propagated back to the last unfolded time step. Interestingly also for initialisations close to $[-0.2, 0.2]$ the error flow stays on a similar level whereas for higher or smaller values it increases and respectively decreases quickly. This saddle point exists for all tested state dimensions, respectively for a slightly shifted interval. A plausible explanation for this, is the transfer function used, the hyperbolic tangent, which stays in the linear range for those values.



(a) Backpropagated error information to the last unfolded time step in relation to the range of the uniform weight initialisation. From left to right the corresponding internal state dimension is $100, 90, 80, 70, 60, 50$, and 40.

(b) Backpropagated error flow to the last unfolded time step in relation to the range of the uniform weight initialisation for the different regarded state dimensions normalised according to equation (3.23).

Figure 3.10: Influence of the initial weight distribution on the backpropagated error flow in RNN using different internal state dimensions.

Figure 3.10(a) also illustrates that the smaller the internal state dimension is the higher the weights should be initialised. This is caused by the fact that with a higher dimension the probability of larger absolute values of the row sum in the weight matrices increases, which leads to larger factors backpropagating the error through the unfolded network. Still, as the curves all run in parallel, there is obviously a connection between the optimal weight initialisation and the internal state dimension of the network.

The following useful conjecture was developed

$$\varrho_2 = \varrho_1 \sqrt{\frac{\dim_1}{\dim_2}} \tag{3.23}$$

where ϱ_i stands for the range of the initialisation interval $[-\varrho_i, \varrho_i]$ and $\dim_i$ for two different internal state dimensions of the RNN ($i = 1, 2$). It is based on considerations about connectivity within a matrix [106] and can be easily confirmed by normalising the results of the initialisation test (fig. 3.10(a)). As expected after normalisation all curves coincide (fig. 3.10(b)), which shows that, for a given problem, there is a general development of the error flow within RNN. Still, independent of the internal state dimension the optimal weight initialisation should be determined for each problem setting and network architecture individually. However, as described, this can always be done by measuring the backpropagated error in the last unfolded time step.

The results correspond to an heuristic for weight initialisation in feedforward neural networks [16]. Thereby a reported rule for an optimal weight initialisation is

$$\varrho = \frac{3}{\sqrt{\dim}} \quad , \tag{3.24}$$

where $\dim$ denotes the dimension of the hidden layer. For the tested RNN this would result in $\varrho = 0.3$, which is slightly larger than the empirically determined value, 0.2. The deviation is probably caused by the different network structures. In contrast to feedforward networks for RNN the number of unfolded time steps m^- has to be taken into account. The larger m^- the greater is the influence of the weight initialisation on the last unfolded state, because the weights factorise the error flow in each time step.

”The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.”

John von Neumann (1903 – 1957)

CHAPTER 4

Recurrent Neural Reinforcement Learning

There have already been a few attempts to combine RL with different kinds of recurrent neural networks, e.g. [3, 4, 62, 80]. Schmidhuber's approach [80] is the first work into this direction. He already applies a form of RNN for learning the reward and the dynamics. Two independent networks are used, one for modelling the dynamics and one for modelling the control. Training is done in parallel or sequential. However, in contrast to the presented dynamical consistent RNN (sec. 3.2.3) the applied networks are fully recurrent but do not offer the same flexibility and adaptation ability. Furthermore, problems with learning have been reported [80], which do not apply to the presented RNN (chap. 3).

Bakker [3, 4] showed remarkable results in combining reinforcement learning with LSTM networks [37]. In his PhD-thesis [3] he developed an integrated approach, which learns the problem's dynamics and the optimal policy at the same time. However, he altered this approach to learning both tasks separately and even showed that this is superior than the original version [4]. Still, LSTM networks do not offer the explicit resemblance, in architecture and method, to RL or respectively MDP like RNN (sec. 4.3). Besides, the presented approach follows the idea of an actor-critic method (sec. 2.4.3), where one network learns the value function and the other one determines the optimal policy. As pointed out, this construction has a couple of drawbacks for the regarded RL problem class (sec. 2.5).

Prokhorov [62] follows a similar idea, but uses recurrent networks close to those applied in this thesis (chap. 3). His approach is also based on the idea of actor-critic (sec. 2.4.3). Due to that, the construction of his neural controller differs essentially. Most important, as in the work of Bakker [4], two recurrent networks are trained, one for the critic and one for the actor. However, it is the only known method that is also applied to an industrial problem [63].

In the following different novel recurrent neural RL approaches based on RNN

(chap. 3) are presented. It starts with a so-called hybrid RNN approach, where RNN are solely used for the system identification of RL problems (sec. 4.1). The method is successfully applied to a partially observable version of the cart-pole problem (sec. 4.2). Subsequent, an enhanced RNN architecture to model and respectively reconstruct (partially observable) Markov decision processes, which improves the hybrid RNN approach, is introduced (sec. 4.3). Finally the recurrent control neural network (RCNN), which combines RL and RNN within one integrated network architecture, is presented (sec. 4.4). Its performance is evaluated on a data-efficient cart-pole (sec. 4.5) and the mountain-car problem (sec. 4.6). At last, the RCNN is further extended to improve its applicability to RL problems of industrial scale (sec. 4.7).

4.1 The Hybrid RNN Approach

As pointed out in the introduction, in technical or economic applications the state $\mathbf{s}_t$ of an RL problem (eq. 1.1) is generally at least partially unknown. Mostly one can only observe a certain number of variables, which might have an influence on the system's dynamics. In contrast, for solving an optimal control problem the knowledge about the dynamics is an essential requirement to estimate the (future) influence of a certain action. Therefore an accurate and robust system, respectively state space, identification is a valuable first step for a real-world application of RL methods (sec. 1.1).

The hybrid RNN approach aims into this direction. It basically consists of a subsequent application of an RNN (sec. 3.2.3) and any standard RL method (sec. 2.4). The idea is quite intuitive. Due to partial observability but also high dimensionality, a couple of RL problems (sec. 2.5) cannot directly be treated by standard RL methods. The hybrid RNN approach now uses an RNN first to transform the state space of the RL problem such that standard RL methods can be applied. As argued in chapter 3, RNN offer an ideal framework to model open dynamical systems. This quality is used to simulate the system development of an RL problem. Hereby one can profit from the description of RL problems by dynamical systems (sec. 1.1) and a structural resemblance between RNN and POMDP (sec. 2.2). The latter aims at the fact that both are state space models. The states of an POMDP can therefore easily be mapped into the architecture of an RNN.

The RNN gets the observable state information $\mathbf{x}_\tau$ as external inputs and targets. Additionally, the last chosen actions $\mathbf{u}_{\tau-1}$ are provided as inputs. Importantly, this is also done in the overshooting part of the network ($\tau > t$) as the actions are necessary to determine the system's dynamics but cannot or should not be learnt by the network. The RNN can then learn the underlying dynamics of

the RL problem within its inner state $\bar{\mathbf{s}}_{t+1}$. Altogether, analogue to equation 3.10, this results in the following set of equations:

$$\begin{aligned}
\bar{\mathbf{s}}_{\tau+1} &= f(A\bar{\mathbf{s}}_\tau + B\begin{pmatrix}\mathbf{x}_\tau \\ \mathbf{u}_{\tau-1}\end{pmatrix} - \theta) \quad \forall \tau \leq t \\
\bar{\mathbf{s}}_{\tau+1} &= f(A\bar{\mathbf{s}}_\tau + B\begin{pmatrix}\bar{\mathbf{x}}_\tau \\ \mathbf{u}_{\tau-1}\end{pmatrix} - \theta) \quad \forall \tau > t \\
\bar{\mathbf{x}}_\tau &= C\bar{\mathbf{s}}_\tau \\
\sum_{t=m_-}^{T-m_+} \sum_{\tau=t-m_-}^{t+m_+} &\|\bar{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2 \rightarrow \min_{A,B,C,\theta}
\end{aligned} \tag{4.1}$$

The corresponding architecture is depicted in figure 4.1. For the illustration of dynamical consistency a matrix $[\mathbf{I}_I\ \mathbf{0}]^T$, where $\mathbf{I}_I$ denotes a fixed identity matrix of dimension I and $\mathbf{0}$ a $K \times I$ dimensional zero matrix, is added. It transfers the predicted observations $\bar{\mathbf{x}}_{\tau+1}$ ($\tau \geq t$) to the input vector without overwriting the observed action $\mathbf{u}_\tau$.

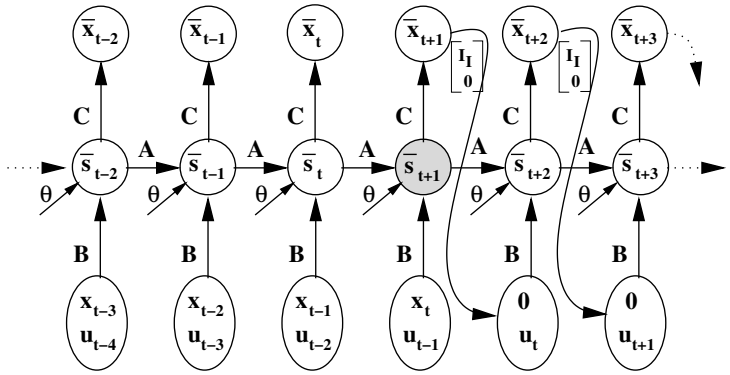


Figure 4.1: RNN architecture for the hybrid RNN approach. Matrix $[\mathbf{I}_I\ \mathbf{0}]^T$ is added for the illustration of dynamical consistency. In the second step the internal state $\bar{\mathbf{s}}_{t+1}$ (shaded) is used as a basis for common RL methods.

The approach can mainly be applied for two different aspects. One can either use the RNN to reduce a problem's dimensionality or to reconstruct a partially observable state space:

(i) A reduction of a problem's dimensionality is done by setting the dimension of the RNN's inner state $\bar{\mathbf{s}}_{t+1}$ to a desired value, which is minimal but sufficient to learn the problem's dynamics. Thus, one can compress the observables' system information to the problem's essential state space. The dimension of

$\bar{\mathbf{s}}_{t+1}$ just has to be large enough to evolve the original system development. Alternatively one can also start with a larger state space and reduce its dimension by incrementally applying node pruning [12, 47]. That way, the RNN provides some kind of feature selection, where in contrast to classical approaches, the features provide across-the-time information.

(ii) A state space reconstruction profits from the advantage that RNN can fairly easy handle partially observable problems as they are, in contrast to most standard RL methods, still able to learn a problem's dynamics although only a small part of the system's information is provided. In short, due to the techniques of unfolding in time and overshooting RNN can develop autonomously the complete system dynamics. They build up a finite memory and learn inter-temporal dependencies out of the available data to compensate the missing information at each time step (sec. 3.2). In doing so they can reconstruct the original state space of the RL environment in their internal state $\bar{\mathbf{s}}_{t+1}$. An application of this approach is given in section 4.2, where it is shown to be a valuable method.

In a second step the developed internal state $\bar{\mathbf{s}}_{t+1}$ of the RNN is used as basis for common RL methods. In other words, having a lower dimensional or reconstructed state space, one can, after a sufficiently fine-gridded discretisation, apply standard and well-known RL algorithms like Q-learning (sec. 2.4.2), prioritised sweeping (sec. 2.4.4), or AHC (sec. 2.4.3) to determine an optimal policy. Similarly one can also use the hybrid RNN approach to extend the applicability of other data-efficient RL methods like least-squares policy-iteration (sec. 2.5) to partially observable problems.

Figure 4.2 illustrates the structure of the hybrid RNN approach, after the RNN has been trained on the observed environmental data $\mathbf{x}_\tau$ and actions $\mathbf{u}_{\tau-1}$. Here, the RNN is used to provide its inner state $\bar{\mathbf{s}}_{t+1}$ as an estimator for the real environmental state $\mathbf{s}_t$. The calculation is done by the RNN's learnt dynamics, which is based on the respective past time observations $\mathbf{x}_\tau$ and chosen actions $\mathbf{u}_{\tau-1}$ ($\tau \leq t$). Note that for this no learning is involved and no overshooting is used. Based on the RNN's inner state $\bar{\mathbf{s}}_{t+1}$ an RL method determines the next action $\bar{\mathbf{u}}_t$. Afterwards the RNN's inputs are shifted by one time step to the past ($\tau \leftarrow \tau-1$) and the chosen action and the resulting next observation $\mathbf{x}_{t+1}$ are given to the network as most recent input. Thus, referring to section 1.1 the RNN fulfils step (i), whereas the RL method is applied for step (ii).

The hybrid RNN approach can be further enhanced by using an extended RNN architecture, where the observables $\mathbf{x}_\tau$ and the control variables $\mathbf{u}_\tau$ are implemented as separate inputs (sec. 4.3).

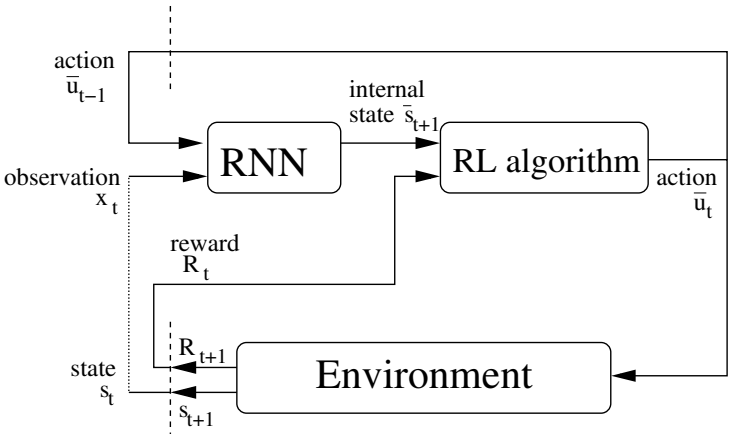


Figure 4.2: Hybrid RNN approach. The trained RNN is applied to minimise or reconstruct the environmental state space $\mathbf{s}_t$ out of the current observation $\mathbf{x}_t$, the last applied action $\bar{\mathbf{u}}_{t-1}$ and its past time information about the system's development. The RNN's inner state $\bar{\mathbf{s}}_{t+1}$ then serves as a basis for an arbitrary RL method, which in turn determines the next action $\bar{\mathbf{u}}_t$. Again, the dashed line indicates the transition to the next time step.

4.2 The Partially Observable Cart-Pole Problem

The classical cart-pole problem [89] consists of a cart, which is able to move on a bounded track and trying to balance a pole on its top. The system is fully defined through four variables ($t = 1, \ldots, T$):

$$\begin{array}{lcl} \chi_t \in \mathbb{R} & := & \text{horizontal cart position} \\ \dot{\chi}_t \in \mathbb{R} & := & \text{horizontal velocity of the cart} \\ \alpha_t \in \mathbb{R} & := & \text{angle between pole and vertical} \\ \dot{\alpha}_t \in \mathbb{R} & := & \text{angular velocity of the pole} \end{array} \tag{4.2}$$

The problem's system dynamics is given by

$$\begin{bmatrix} M+m & ml\cos\alpha_t \\ ml\cos\alpha_t & \frac{3}{4}ml^2 \end{bmatrix} \begin{bmatrix} \ddot{\chi}_t \\ \ddot{\alpha}_t \end{bmatrix} - \begin{bmatrix} ml\dot{\alpha}_t{}^2 \sin\alpha_t \\ mgl\sin\alpha_t \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix}$$

where $M \in \mathbb{R}^+$ and $m \in \mathbb{R}^+$ are the masses of the cart and pole respectively, $l \in \mathbb{R}^+$ is half the length of the pole, $g \in \mathbb{R}^+$ the acceleration due to gravity and $F \in \mathbb{R}$ the force applied to the cart. Here, the following setting was used [89]: $\chi_t \in [-2.4, 2.4]$ and $\alpha_t \in [-12°, 12°]$ $\forall t = 1, \ldots, T$, $M := 1.0$, $m := 0.1$, $l := 0.5$, $g := 9.8$, and $F = 10.0$. The system is illustrated in figure 4.3.

The goal is to balance the pole for a preferably long sequence of time steps without moving out of the limits. Possible actions are to push the cart left or

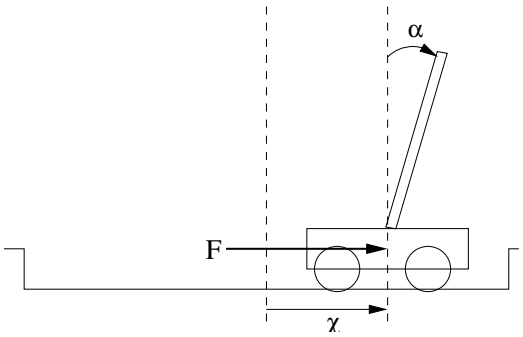


Figure 4.3: The cart-pole problem [53].

right with the constant force F. The pole tilts when its angle α_t is larger than 12 degrees. Either then or when the cart hits one of the boundaries (± 2.4) the system is punished with a negative reward of one. In all other cases the reward is zero.

The problem has been extensively studied in control and RL theory and serves as a standard benchmark, because it is easy to understand and also quite representative for related questions. The classical problem has been completely solved in the past. Sutton and Barto [89] for example demonstrated that the pole can be balanced for an arbitrary number of time steps.

Still, there exist a couple of variations, which are generally more challenging. Gomez for example solved the problem with two poles [25], whereas Gomez and Miikkulainen considered a two dimensional cart [26]. Besides, often the original problem is regarded as only partially observable [2, 53]. A good summary of the different problem classes is given in [99].

Nevertheless so far nobody tried to reduce the observability to only one single variable. When the system was studied as partially observable, one usually omitted the two velocities, $\dot{\chi}_t$ and $\dot{\alpha}_t$, i.e. only the cart's position and the angle between the pole and the vertical were given as inputs [2, 25, 53]. Solving this problem is not very challenging because the model or algorithm just needs the memory of one past time step to calculate the missing information.

While the following experiment is aimed at fully profiting from the advantages of RNN reconstructing a partially observable state space (sec. 4.1), only the horizontal position of the cart, χ_t, is given as an observable. All other information is absolutely unknown to the system.

In an extension the problem is even further complicated by adding noise to the only observable variable χ_t. This strengthens the requirement that the hybrid RNN approach cannot absolutely rely on the single information that it receives about the cart-pole problem's environment, but has to extract the true underlying dynamics.

4.2.1 Model Description

To solve the described partially observable cart-pole problem the hybrid RNN approach was applied. According to its description (sec. 4.1), first an RNN (eq. 4.1) was used to develop the full dynamics of the cart-pole system. Here, the observable environmental information consisted of the horizontal cart position χ_t as well as the first and second discrete differences of it. It was given to the RNN as part of the input and as target $\mathbf{x}_\tau \in \mathbb{R}^3$. Additionally, the input contained the agent's last action $\mathbf{u}_{\tau-1} \in \mathbb{R}$. No other information was observable by the model. The internal state space $\bar{\mathbf{s}}_t \in \mathbb{R}^4$ was limited to four neurons. Thus, it was intended that the RNN reconstructs the complete but only partially observable environment (eq. 4.2) in its internal state space. The network was respectively unfolded ten time steps into past and future. The results had shown that this memory length was sufficient to identify the dynamics. To make the network independent from the last unfolded state cleaning noise was used as a start initialisation (sec. 3.5.2). The hyperbolic tangent was implemented as activation function f and the RNN was trained with the shared weight extended backpropagation (sec. 3.4.1) and pattern-by-pattern learning (sec. 3.4.2.1) until a minimum error between output $\bar{\mathbf{x}}_t$ and target $\mathbf{x}_t$ is achieved.

In the second step the adaptive heuristic critic (AHC) algorithm (sec. 2.4.3), which has shown competitive results on the standard cart-pole problem setting, was applied on the developed internal state $\bar{\mathbf{s}}_{t+1}$ of the RNN. Note that, due to the unfolding in time of the RNN, the algorithm had to be started with an already filled lag structure. Otherwise the first ten steps would be uncontrolled and consequently there would be a high probability that the algorithm is faced with an unstable pole in its first learning step.

4.2.2 Results

Several different data sets were used to train the described RNN (sec. 4.2.1). As a first result it was confirmed that the number of observed transitions is more important than the number of training epochs. The more different information about the single input variable the network experienced the more it was able to reconstruct the original (complete) state space.

As a verification of if and how well the RNN could identify the underlying dynamics out of the single observable variable, the original and the reconstructed state vectors were analysed and compared. The result is shown in figure 4.4. The four plots show the correlation between the original state space variables of the environment, $\chi_t, \dot{\chi}_t, \alpha_t, \dot{\alpha}_t$, (eq. 4.2) and the best linear combination of the reconstructed state space variables $(\bar{s}_{t+1})_1, \ldots, (\bar{s}_{t+1})_4$ and their squares $(\bar{s}_{t+1})_1^2, \ldots, (\bar{s}_{t+1})_4^2$ in each case. It turned out that the correlation for each state

space variable was very high. This demonstrates the reconstruction quality of the RNN and underlines the use of the hybrid RNN approach for partially observable RL problems.

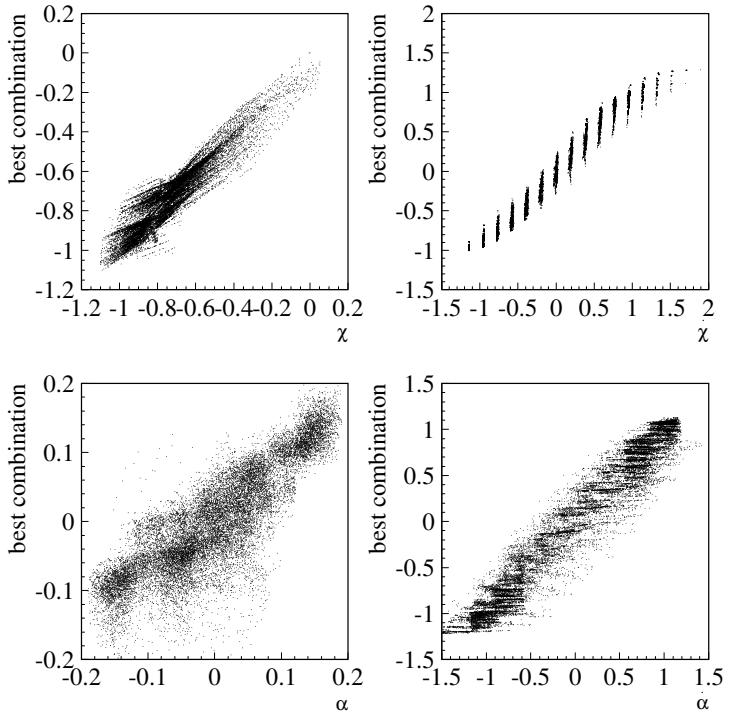
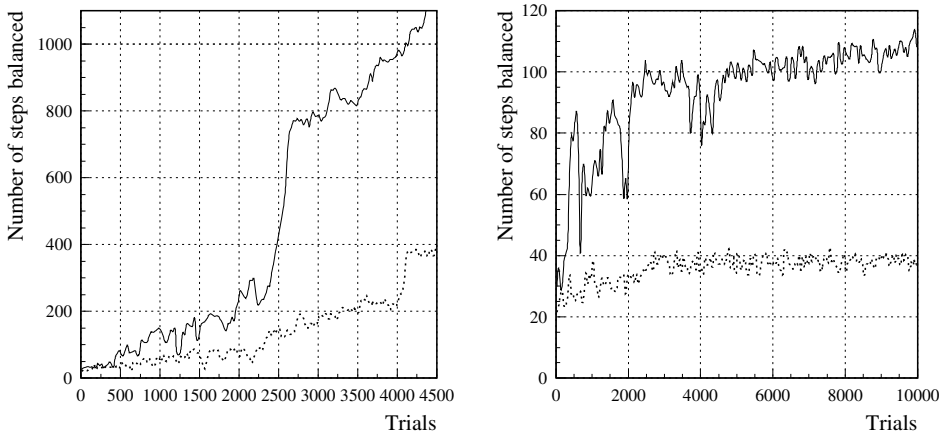


Figure 4.4: Correlation between the best quadratic combination of the reconstructed state space variables $(\bar{s}_{t+1})_1, \ldots, (\bar{s}_{t+1})_4$ of the RNN and the original ones (eq. 4.2).

The results of the hybrid RNN approach were compared to a direct application of the AHC algorithm, which means without using an RNN in the first step. In both cases the discretisation of the state space that yielded the best results was taken. Figure 4.5(a) plots the achieved number of steps the pole could be balanced to the number of trials. The training was stopped when the first method was able to balance the pole for a minimum of 1000 steps. The result reveals the advantage of the hybrid RNN approach as it outperforms a direct application of the AHC algorithm by far.

The better performance of the hybrid RNN approach became even more obvious when a Gaussian noise was added to the single observation χ_t. Already for a 1% noise level a direct application of the AHC algorithm failed almost completely to learn the task. In contrast, the hybrid RNN approach was, for all tested noise levels, able to balance the pole for at least more than a hundred time steps

(fig. 4.5(b)). This result well demonstrates that the RNN is still able to identify and reconstruct the original state space of the environment although the only observable information is covered by noise.



(a) Without Noise (b) With Noise

Figure 4.5: Comparison of the performance in the partially observable cart-pole problem of the hybrid RNN approach (solid) to a direct application of the AHC algorithm (dotted). The hybrid RNN approach clearly outperforms the direct application of the AHC. With a 1% noise level on the single observable variable χ_t it is able to balance the pole for at least a hundred time steps whereas the direct application of the AHC fails almost completely to learn the task. The curves have been averaged over 50 trials.

4.3 Markovian State Space Reconstruction by RNN

The basic hybrid RNN approach (sec. 4.1) applies a standard RNN to identify and respectively reconstruct the RL problem's state space. The following RNN resumes this idea by adapting its structure to the task of reconstructing the system's (minimal) Markovian state space. Here, one can profit from the described extension ability of RNN (chap. 3).

Modelling and respectively reconstructing higher-order POMDP (sec. 2.1) with RNN (sec. 3.2) follows the idea of explicitly mapping the process's dynamics by a high-dimensional non-linear system equation. Similar to equation 4.1 the RNN is therefore constructed such that it receives the external observations

$\mathbf{x}_\tau$ ($\tau = 1, \ldots, T$) of a POMDP (sec. 2.2) as inputs and targets. However, now the actions $\mathbf{u}_\tau$ are given to the network as separate inputs (fig. 4.6). In doing so one gets, analogue to a POMDP, the sequence of the observable states $\mathbf{x}_\tau$ and the subsequent actions $\mathbf{u}_\tau$ on the input side of the RNN. Moreover, also with regard to the later extension to the recurrent control neural network (RCNN) (sec. 4.4), an additional internal state $\hat{\mathbf{s}}_\tau$ is included, which adapts the role of the approximately Markovian state. It allows to explicitly reconstruct the RL problem's state $\mathbf{s}_\tau$ out of the observables $\mathbf{x}_\tau$ (respectively $\bar{\mathbf{x}}_\tau$) and the RNN's inner state $\bar{\mathbf{s}}_\tau$, before applying a new action $\bar{\mathbf{u}}_\tau$. The vector $\bar{\mathbf{s}}_\tau$ represents as hitherto the internal state of the RNN determining the target, i.e., the observations, $\mathbf{x}_\tau$. This results in the following optimisation problem:

$$\begin{aligned} \bar{\mathbf{s}}_{\tau+1} &= f(\mathbf{I}_{\bar{J}}\hat{\mathbf{s}}_\tau + D\mathbf{u}_\tau - \theta) \\ \bar{\mathbf{x}}_\tau &= C\bar{\mathbf{s}}_\tau \\ \text{with} \quad \hat{\mathbf{s}}_\tau &= \begin{cases} A\bar{\mathbf{s}}_\tau + B\mathbf{x}_\tau & \forall \tau \leq t \\ A\bar{\mathbf{s}}_\tau + B\bar{\mathbf{x}}_\tau & \forall \tau > t \end{cases} \\ \sum_{t=m_-}^{T-m_+} \sum_{\tau=t-m_-}^{t+m_+} \|\bar{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2 &\rightarrow \min_{A,B,C,D,\theta} \end{aligned} \tag{4.3}$$

Here, analogue to equation 3.5, the state transition equation $\bar{\mathbf{s}}_{\tau+1} \in \mathbb{R}^{\bar{J}}$ is a non-linear transformation of the previous approximately Markovian state $\hat{\mathbf{s}}_\tau \in \mathbb{R}^{\bar{J}}$, the actions $\mathbf{u}_\tau \in \mathbb{R}^K$ using weight matrix $D \in \mathbb{R}^{K \times \bar{J}}$, a fixed identity matrix $\mathbf{I}_{\bar{J}}$ of dimension $\bar{J}$, and a bias $\theta \in \mathbb{R}^{\bar{J}}$. As pointed out, the approximately Markovian state $\hat{\mathbf{s}}_\tau$ aggregates the information from the internal state $\bar{\mathbf{s}}_\tau$ and respectively the external observation, $\mathbf{x}_\tau \in \mathbb{R}^I$, or the network's own prediction for it, $\bar{\mathbf{x}}_\tau \in \mathbb{R}^I$, applying weight matrices $A \in \mathbb{R}^{\bar{J} \times \bar{J}}$ and $B \in \mathbb{R}^{I \times \bar{J}}$. The expected next observation of the POMDP, $\bar{\mathbf{x}}_{\tau+1} \in \mathbb{R}^I$, is computed from the previous internal state $\bar{\mathbf{s}}_{\tau+1}$ employing matrix $C \in \mathbb{R}^{\bar{J} \times I}$. As hitherto, f denotes an arbitrary non-linear activation function. Analogue to the basic hybrid RNN approach (sec. 4.1), the actions $\mathbf{u}_\tau$ are also given to the RNN as future inputs ($\tau > t$) because they directly influence the POMDP's dynamics but cannot or should not be learnt by the network. Besides, the approximated state space of the RNN (eq. 4.3) in general does not have the same dimensionality as the one of the original POMDP (sec. 2.2). It basically depends on the order and complexity of the respective POMDP as well as the desired accuracy. Again, the RNN is trained with the shared weights extension of the backpropagation algorithm (sec. 3.4.1) and a suitable learning algorithm (sec. 3.4.2). Figure 4.6 depicts the resulting RNN architecture.

As shown in section 3.3, RNN are able to approximate any deterministic open dynamical system arbitrarily precisely. Although this does not necessarily apply

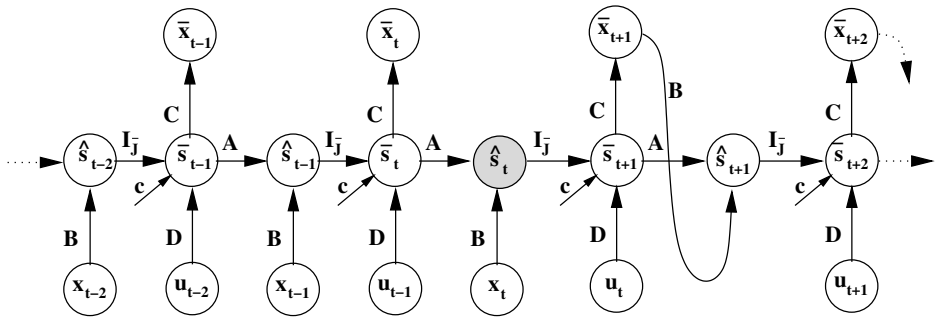


Figure 4.6: RNN architecture for modelling POMDP: Observations $\mathbf{x}_\tau$ and actions $\mathbf{u}_\tau$ are modelled as separate inputs. The additional state $\hat{\mathbf{s}}_\tau$ adapts the role of an approximately Markovian state. It allows to explicitly reconstruct the RL problem's state $\mathbf{s}_\tau$ out of the observables $\mathbf{x}_\tau$ (respectively $\bar{\mathbf{x}}_\tau$) and the networks inner state $\bar{\mathbf{s}}_\tau$. Thus, regarding the hybrid RNN approach $\hat{\mathbf{s}}_t$ (shaded) can be used in second step as a basis for an arbitrary RL method.

for stochastic systems, there is no true restriction. The construction of the state $\hat{\mathbf{s}}_\tau$ can just be seen as the transformation into an appropriate Markovian feature space, which is built from past and present observations. In the deterministic case, the feature space is identical to the perfect description of all information determining the future, while in the general stochastic case the state $\hat{\mathbf{s}}_\tau$ has to be designed in order to forecast the expected future. Additionally, it has been shown that under slightly restrictive conditions the Markov property can indeed be reconstructed by knowing the expected future states while the state space is observed partially [82].

Summarising, the RNN (eq. 4.3) is used to identify the system's (minimal) approximately Markovian state space out of the observed data. For this, the dimension of the Markovian state $\hat{\mathbf{s}}_\tau$ of the RNN can be set to a desired value, which is minimal but sufficient to learn the problem's dynamics. In doing so, one can compress the observables' system information to the problem's essential state space. The dimension of $\hat{\mathbf{s}}_\tau$ just has to be large enough to evolve the original system development.

Analogue to the description in section 4.1 in a second step then any standard RL method (sec. 2.4) can be applied. However, those are now based on the Markovian state $\hat{\mathbf{s}}_t$, which has been especially inserted for the reconstruction of the POMDP's state space. An application of this approach to gas turbine control is given in chapter 5.

4.4 The Recurrent Control Neural Network

The recurrent control neural network (RCNN) has been developed for the purpose to identify and control the dynamics of an RL or optimal control problem (sec. 1.1) directly within one integrated network. Thus, it rounds off the hybrid RNN approach (sec. 4.1) in the sense that the RL problem (steps (i)+(ii)) can now be completely solved by a recurrent neural network.

The principal architecture of the RCNN is based on the RNN for the Markovian state space reconstruction (sec. 4.3). It is extended by an additional control network and an output layer, which incorporates the reward function. Overall its integrated structure follows the idea of solving the complete optimal control problem within one network, i.e., the system identification (step (i)) as described in section 3.2 (and respectively section 4.3) and learning the optimal policy (step (ii)). Here, the policy is determined directly by a maximisation of the finite sum of expected future rewards without calculating any value function. In this regard the RCNN also has some similarity to policy gradient methods (sec. 2.4.5).

The additional and integrated control network has the form of a three-layered FFNN (sec. 3.1). Despite other (more extensive) topologies would be possible, this already allows to model any arbitrary control function (sec. 3.3.1). As one wants to predict the optimal (future) actions $\mathbf{u}_\tau \in \mathbb{R}^K$, the control network is only applied in the present and overshooting part of the RCNN ($\tau \geq t$) (fig. 4.7, dashed part). In the past unfolding ($\tau < t$) the RCNN is provided with the last actions taken. The control network uses the values of the determined approximately Markovian state $\hat{\mathbf{s}}_\tau$, which combines the information of the inner state $\bar{\mathbf{s}}_\tau$ and the environmental observables $\mathbf{x}_\tau$ (or respectively $\bar{\mathbf{x}}_\tau$) as inputs. As an output it determines the next action or control variables $\bar{\mathbf{u}}_\tau$ ($\tau \geq t$). Putting this into equations the control network has the form ($\forall \tau \geq t$)

$$\bar{\mathbf{u}}_\tau \quad = \quad f_u(F f_c(E\hat{\mathbf{s}}_\tau - \mathbf{b})) \tag{4.4}$$

where $E \in \mathbb{R}^{H \times \bar{J}}$ and $F \in \mathbb{R}^{K \times H}$, with $H \in \mathbb{N}$ as the number of hidden neurons of the control network, are weight matrices, $\mathbf{b} \in \mathbb{R}^H$ is a bias and f_c an arbitrary activation function. Additionally, f_u denotes a problem-specific, component-wise applied activation function, which can be used to scale or limit the network's action space. The hidden state (fig. 4.7) of the control network is denoted by $\mathbf{h}_\tau \in \mathbb{R}^H$.

The RCNN has to fulfil two different tasks (sec. 1.1), the identification of the problem's dynamics (step (i)) and the optimal control (step (ii)), and is hence trained in two successive steps. For both steps the training is done offline on the basis of previous observations. This again distinguishes the approach from other work on RL and recurrent neural networks, e.g. [3], where one usually tries a combined learning of both tasks in one step.

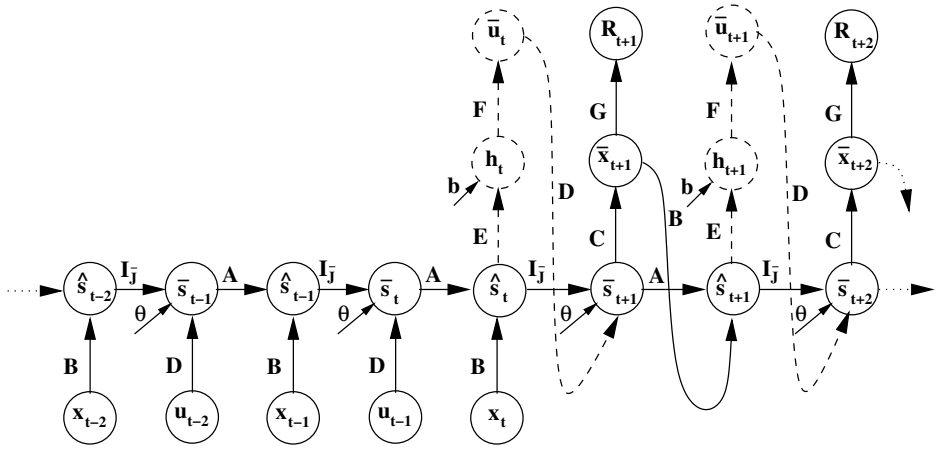


Figure 4.7: Recurrent Control Neural Network Architecture of step two: The control network (dashed) determines the policy by adjusting the weight matrices E, F and the bias $\mathbf{b}$ according to the gradient flow from the reward cluster R_τ ($\tau > t$). Matrices A, B, C, and D, and the bias θ, which code the dynamics, are fixed.

In the first step the RCNN is limited to the identification and modelling of the dynamics of the underlying POMDP. It is consequently reduced to an RNN reconstructing a Markovian state space (sec. 4.3). Hence, the optimisation task of step one takes on the following form with the variables defined as in equation 4.3:

$$\begin{aligned} \bar{\mathbf{s}}_{\tau+1} &= f(\mathbf{I}_{\bar{j}}\hat{\mathbf{s}}_\tau + D\mathbf{u}_\tau - \theta) \\ \bar{\mathbf{x}}_\tau &= C\bar{\mathbf{s}}_\tau \\ \text{with} \quad \hat{\mathbf{s}}_\tau &= \begin{cases} A\bar{\mathbf{s}}_\tau + B\mathbf{x}_\tau & \forall \tau \leq t \\ A\bar{\mathbf{s}}_\tau + B\bar{\mathbf{x}}_\tau & \forall \tau > t \end{cases} \end{aligned} \tag{4.5}$$

$$\sum_{t=m_-}^{T-m_+} \sum_{\tau=t-m_-}^{t+m_+} \|\bar{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2 \rightarrow \min_{A,B,C,D,\theta}$$

In the second step all connections coding the dynamics and learnt in the first step, in particular matrices A, B, C, and D and the bias θ, get fixed, while the integrated control network with the matrices E and F and the bias $\mathbf{b}$ is activated (fig. 4.7, dashed part). These are also the only tunable parameters in this training step. Besides that, as the RCNN's task is now to learn the optimal policy it does not get the future actions as external inputs in this step (fig. 4.7). Still, in the past unfolding ($\tau < t$) the RCNN is, as in step one, still provided with the actions $\mathbf{u}_\tau$ of the observed training data. Furthermore, in the past unfolding ($\tau < t$) the

output-clusters are deactivated, because they are only needed for the identification of the system dynamics. However, in the present and future part ($\tau \geq t$) of the network the error-function (eq. 4.5) of the output clusters gets replaced by the reward function. Architecturally this is realised by additional reward clusters R_τ, which are connected to the output cluster by a problem specific, on the reward function R (sec. 1.1) dependent, and fixed matrix G as well as a possible activation function f_r within the output clusters $\bar{\mathbf{x}}_\tau$ (fig. 4.7). In other words the RCNN maps the reward function R of the underlying RL problem by coding it in a neural architecture, which, as in the case of gas turbine control (chap. 5), often also requires some additional, fixed connected, clusters. Here, also a discount factor γ (eq. 1.2) could be incorporated but is generally omitted due to the finiteness of the future unfolding, i.e., the calculation of a finite sum of future rewards. It is further possible to learn the reward function from the observations, which is especially of interest in cases where R is not known or incompletely specified in the problem setting. This can be realised by an additional three-layered FFNN.[1]

The weights of the control network are only adapted according to the back-propagated error from the reward clusters R_τ ($\tau > t$). This follows the idea that in the second step one wants to learn a policy that maximises the finite sum of expected future rewards given the system dynamics modelled in step one (eq. 4.5). Note that in doing so the learning algorithm changes from a descriptive to a normative error function.

Summarising, step two can be represented by the following set of equations (eq. 4.6). Here, bold capital letters stand for fixed matrices, which are not learnt in this step.

$$\begin{aligned}
\bar{\mathbf{s}}_{\tau+1} &= \begin{cases} f(\mathbf{I}_J \hat{\mathbf{s}}_\tau + \mathbf{D}\mathbf{u}_\tau - \theta) & \forall \tau < t \\ f(\mathbf{I}_J \hat{\mathbf{s}}_\tau + \mathbf{D}\bar{\mathbf{u}}_\tau - \theta) & \forall \tau \geq t \end{cases} \\
R_\tau &= \mathbf{G} f_r(\mathbf{C}\bar{\mathbf{s}}_\tau), \quad \forall \tau > t \\
\text{with} \quad \bar{\mathbf{u}}_\tau &= f_u(F f_c(E\hat{\mathbf{s}}_\tau - \mathbf{b})) \quad \forall \tau \geq t \\
\text{and} \quad \hat{\mathbf{s}}_\tau &= \begin{cases} \mathbf{A}\bar{\mathbf{s}}_\tau + \mathbf{B}\mathbf{x}_\tau & \forall \tau \leq t \\ \mathbf{A}\bar{\mathbf{s}}_\tau + \mathbf{B}\bar{\mathbf{x}}_\tau & \forall \tau > t \end{cases} \\
\sum_{t=m_-}^{T-m_+} \sum_{\tau>t}^{t+m_+} R_\tau &\rightarrow \max_{E,F,b}
\end{aligned} \tag{4.6}$$

The architecture of the RCNN in the second step, i.e. during learning of the optimal control, is depicted in figure 4.7.

[1] By an additional connector from $\mathbf{u}_\tau$ to $R_{\tau+1}$ one can also easily incorporate the applied action into the reward function (sec. 2.1).

	Recurrent control neural network
Technology	Neural networks
Problem class	High-dimensionality, partial observability, continuous state and action space, data-efficiency
Philosophy	Identification of system dynamics and optimal policy by an integrated RNN
Data-efficiency	Through batch approximation of dynamics and analytical incorporation of a reward function
Prior knowledge	Reward function and influences of actions on a subset of the state space
Biasedness	Through system identification
Task of RNN	Identification of the system dynamics
Task of control network	Maximising sum of future rewards by virtual Monte-Carlo policy gradient

Table 4.1: Summarised features of the recurrent control neural network.

In both steps the RCNN is trained on the identical set of training patterns T and with the shared weight extended backpropagation (sec. 3.4.1). Concerning the second step this means in a metaphoric sense that by backpropagating the loss of the reward function R_τ the algorithm fulfils the task of transferring the reward back to the agent. In either case, an optimal system identification is an essential part of the RCNN as it forms the basis for learning the optimal policy in the second step. Consequently, in both steps the RCNN must be operated until a sufficient generalisation performance is achieved.

As already pointed out, the second step has some connection to policy gradient methods (sec. 2.4.5). Analogue to those the RCNN makes no use of the value function. It rather searches directly within in the policy space and uses the reward's gradient for its policy improvement. In fact, the RCNN can be seen as a virtual Monte-Carlo policy gradient approach as the value of the current policy is determined virtually within the unfolded network and improved according to the gradient of the reward function. On this basis an explicit Monte-Carlo estimation [52], which would require a generation of new data-samples, is avoided. This increases the RCNN's data-efficiency.

Table 4.1 outlines the main characteristics and features of the RCNN. Summarising, the RCNN ideally combines the advantages of an RNN with dynamically consistent overshooting for identifying the problem's dynamics and an FFNN for learning the optimal policy. In doing so, one can benefit from a high approximation accuracy and therefore control extensive dynamics in a very data-efficient way. Besides that, one can easily scale into high dimensions or recon-

struct a partially observable environment (sec. 4.3). Furthermore, by construction of the RCNN it can well handle continuous state and action spaces.

A further advantage of the RCNN is the inherent embedding of an analytically given reward function. This allows in the second step to maximise the sum of expected future rewards without any statistical bias. However, a statistical bias might be brought into the network by learning the dynamics in the first step. For the same reason it is structurally difficult to solve stochastic problems. Still, the RCNN has the possibility to deal with stochasticity by using a higher-dimensional and more extensive deterministic dynamics.

4.5 The Data-Efficient Cart-Pole Problem

The problem setting has already been described in section 4.2. As pointed out, in its classical form it has been completely solved in the past, e.g. [89]. Still all successful methods need quite a large number of training patterns to find a solution. In contrast, as described in section 2.5, for real-world applications training data is in general very limited. Consequently, methods that require less training data to solve the problem, i.e., which are more data-efficient, are preferable. In the following experiment therefore a special focus is put on data-efficiency and it is shown that the RCNN achieves outstanding results. However, in contrast to the experiment in section 4.2 the system is now fully observable. Similar tests have been reported in [68], but with a slightly different dynamics and an extended action space.

4.5.1 Model Description

An RCNN as described in section 4.4 was used with an unfolding of 10 steps into the past and 30 into the future. This gave the network both, a memory length, which was sufficient to identify the dynamics, and an overshooting length, which enabled it to predict the consequences of its chosen actions. To make the network independent from the last unfolded state, start noise (sec. 3.5) was used as start initialisation. The internal state dimension, $\dim(\mathbf{s})$, was set to 20 and the hidden state of the control network, $\dim(\mathbf{h})$, to 40 neurons. These dimensions were effectual to generate stable results in terms of system identification and learning the optimal policy. Larger networks in general only require more computational time. Furthermore, the hyperbolic tangent was implemented as activation functions f, f_c and f_u. The latter limits the action space of the RCNN to $(-1, 1)$ (eq. 4.4).

For training the RCNN, data of different set sizes was generated where the actions were chosen randomly. Here, it was varied from the standard setting as the originally episodic task was transformed into a continuous one by keeping the

pole or cart at their limits instead of starting a new episode when the pole falls or the cart hits one of the boundaries. Hence, a reward function of the form

$$R = -\sum_{t=m_-}^{T-m_+} \sum_{\tau \geq t}^{t+m_+} \left[(g\chi_\tau)^2 + (\alpha_\tau)^2 \right] \tag{4.7}$$

was used, where g is a scaling factor, which balances the error values of the two variables. In the experiment g was set to 0.1. According to this and recalling that the (predicted) observations are of the form $\bar{\mathbf{x}}_\tau = [\chi_\tau \dot{\chi}_\tau \alpha_\tau \dot{\alpha}_\tau]^T$, matrix G takes on the form

$$G = \begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

the activation function f_r in the outputs $\bar{\mathbf{x}}_\tau$ is set to identity, and the clusters R_τ get a squared error function with constant targets of zero.

The adaption made the time series more applicable for the RCNN, in particular learning with backpropagation (sec. 3.4.1), but did not simplify the problem; especially as the generated data was only used for training. The learnt policy was later tested on the original system and consequently had to cope with the slightly different setting. Here also the continuous action space of the network got rediscretised to -1 and 1.

4.5.2 Results

The RCNN was trained with different amounts of training data. The learnt policy was then tested on the original dynamics of the cart-pole problem where the number of steps $N \in \mathbb{N}$, which it was able to balance the pole, was measured. Respectively three data sets were used with $300, 1000, 3000$, 10000, 30000, and 100000 training patterns. For each set the median of the performance over 100 different random start initialisations of the cart and pole was taken during testing. The results are given for each set size as the median and average over the values of the respectively three different training data sets (tab. 4.2). The maximum number of steps balanced was set to $\max = 100000$, which was considered as sufficient to demonstrate that the policy has solved the problem.

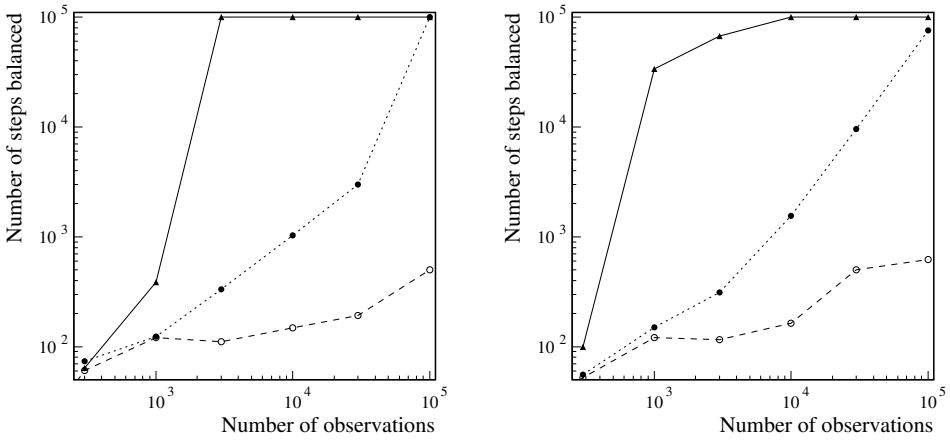
The results were compared to the adaptive heuristic critic (AHC) algorithm (sec. 2.4.3), which shows competitive results on the cart-pole problem. As a second benchmark served (table-based) Q-learning (sec. 2.4.2), which is one of the commonly used RL methods (sec. 2.4). In contrast to the RCNN for both algorithms the standard setting of the cart-pole problem was used, because their application to the modified one (eq. 4.7) produced inferior results.

The results (tab. 4.2) clearly indicate that the RCNN can solve the cart-pole problem very data-efficiently. With only 1000 training patterns the average num-

# of	RCNN		AHC		Q-learning	
obs	median	average	median	average	median	average
300	61	100	74	56	61	52
1000	387	33573	124	150	121	121
3000	max	66912	334	312	111	116
10000	max	max	1033	1554	148	163
30000	max	max	2988	9546	193	501
100000	max	max	max	75393	503	624

Table 4.2: Median and average number of steps the pole was balanced by the RCNN, the AHC, and the Q-learning policy given different numbers of observations (obs).

ber of steps balanced is very high. On one of the tested training data sets with 1000 observations even an optimal policy was learnt. With already 10000 observations the maximum number of steps balanced was achieved on the basis of all tested three data sets. In comparison, the AHC needed at least a 100000 observations for finding a satisfying solution. Q-learning required even more observations, as it still failed to balance the pole with the maximum number of observations tested. Figures 4.8(a) and 4.8(b) illustrate the results for the median and average number of steps balanced.



(a) Median (b) Average

Figure 4.8: Numbers of steps balanced with respect to the number of observations taken the median over the different tested data sets. The RCNN (solid) clearly outperformed the AHC (dotted) and Q-learning (dashed).

To examine the stability of the RCNN policy a uniform noise was put on the force F of the action (sec. 4.2). The task was particularly difficult because the network had not seen any noise during training but its policy had to cope with it during the test on the original dynamics. The median and average of the performance over a hundred different random start initialisations of the cart and pole were taken. Table 4.3 shows the results for different noise levels on an RCNN policy trained with 10000 observations. It demonstrates that even with a noise level of a 100% the RCNN policy balanced the pole in median for the maximum number of steps. Note that a noise level of more than 100% means that the cart can be pushed into the reverse direction of the one, intended by the policy. This also explains the sharp decrease in performance after increasing the noise to more than 100%. Figure 4.9 illustrates the robust performance of the RCNN policy.

noise level on F	# of steps balanced	
	median	average
10%	max	max
20%	max	max
30%	max	99953
40%	max	97337
50%	max	98019
60%	max	96937
70%	max	92886
80%	max	88181
90%	max	84191
100%	max	74554
110%	55154	53775
120%	16519	27730
130%	8238	12476
140%	2961	5294
150%	1865	2863
160%	1008	1503
170%	557	939
180%	173	555
190%	76	344
200%	83	242

Table 4.3: Median and average number of steps balanced with different noise levels on the force F by an RCNN trained with 10000 observations. Even with a noise of a 100% the RCNN was able to balance the pole the maximum number of time steps.

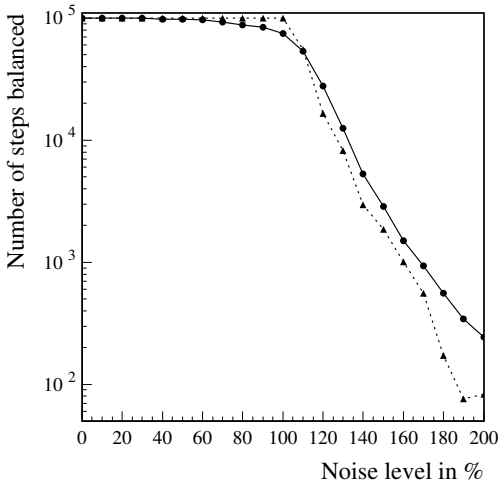


Figure 4.9: Median (solid) and average (dotted) number of steps balanced with different noise levels on the force F by an RCNN trained with 10000 observations.

4.6 The Mountain Car Problem

The application of the RCNN to the cart-pole problem (sec. 4.5) showed its data-efficiency. To demonstrate that the RCNN is able to achieve an optimal policy it is further tested on the mountain car problem, which is fully described in [89]. Its objective is to reach the top of a hill with an underpowered car by gaining kinetic energy through driving forward and backward . The car's continuous state consists of a position $p \in \mathbb{R}$ and a velocity $v \in \mathbb{R}$. There are three possible actions: wait, full power forward and backward, which is equivalent to the application of zero, a fixed positive or negative force $F \in \mathbb{R}$. The system dynamics of the problem is given by the following set of equations

$$\begin{aligned} p_{t+1} &= p_t + v_t \\ v_{t+1} &= v_t + 0.001F - 0.0025\cos(3p_t) \end{aligned}$$

whereby in the used setting $p_t \in [-1.2, 0.5]$, $v_t \in [-0.07, 0.07]$ $\forall t$ and $F := \{-1, 0, 1\}$. In the case p_t and v_t hit one of their boundaries, their values remain unchanged within their domain. The reward is one when the car has reached the top of the right hill ($p_t = 0.5$) and zero otherwise [89]. A trivial near-optimal policy, as reported in [87], is to drive the car always full power in the direction of the car's current velocity v_t. The system is illustrated in figure 4.10.

Two different settings of the outlined problem were regarded, the standard [89] and a slightly simplified one. In the latter so-called meta-actions or options

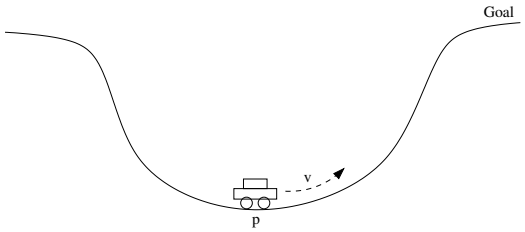


Figure 4.10: The mountain car problem.

[61] were used, which let the agent only take a decision in every fifth time step. This allows the car to travel longer distances in between taking actions. Consequently the car can reach the top of the hill with less decision steps than in the standard setting. For each setting 100000 observations were allowed for training and the learnt policy was afterwards tested on the respective simulated dynamics. Note that, like on the cart-pole problem (sec. 4.5), for the latter the continuous actions determined by the RCNN had to be re-discretised, which can be seen as an additional difficulty. In the simplified setting the training set for the RCNN was created with random actions. For the standard version ε-greedy prioritised sweeping [89] (with $\varepsilon = 0.5$) was pre-applied to obtain a representative training set because random actions never reached the top of the hill.

4.6.1 Model Description

The used RCNN was similar to the one applied to the data-efficient cart-pole problem (sec. 4.5). It is ten time steps unfolded into the past. For the standard setting the future unfolding counted 300 and for the simplified one 50 time steps, which allowed the network to see at least one goal state within its (finite) future horizon. Again, start noise (sec. 3.5.2) was used to make the network independent from the initial unfolded state. Furthermore, it was again $\dim(\mathbf{s}) = 20$ and $\dim(\mathbf{h}) = 40$. The problem was supposed to be fully observable, which implied that the environmental state information p_τ and v_τ were given to the network as inputs and targets $\mathbf{x}_\tau$. Anew, the hyperbolic tangent was implemented as activation functions f, f_c and f_u, which also limited the RCNN's action space to $(-1, 1)$ (eq. 4.4).

According to the problem setting the following reward function was used:

$$R = \sum_{t=m_-}^{T-m_+} \sum_{\tau \geq t}^{t+m_+} \text{logistic}_{10}(p_\tau - 0.5)$$

where $\text{logistic}_{10}(x) = \frac{1}{1+e^{-10x}}$ is a steep logistic function, which is close to a threshold-function but still differentiable. This implies that it approximately is

one for $p_\tau > 0.5$ and zero otherwise. Due to that, in the equations of the RCNN (eq. 4.6) matrix G was set to $G := [1\ 0]$ and the activation function f_r within the outputs corresponded to the described logistic function. Besides, a bias of the value -0.5 was added within the reward clusters R_τ, where further an error function, which maximises the output, was implemented.

4.6.2 Results

The results were compared to the described trivial near-optimal policy [87], standard PS (sec. 2.4.4) and the minimum number of decision steps determined by manual tuning. Table 4.4 summarises the results for the two settings. It shows that the RCNN is able to learn a potentially optimal policy as it was even as good as the manually tuned one. It also outperformed the best results on the problem reported in [1].

	RCNN	PS	Trivial Near-Optimal	Potentially Optimal
Standard	104	144	125	104
Simplified	21	27	26	21

Table 4.4: Number of decision steps needed by the RCNN, the PS, the trivial near-optimal [87], and the manually tuned potentially optimal policy to drive the car up the hill.

4.7 Extended Recurrent Control Neural Network

In the course of its application to gas turbine control (chap. 5) the RCNN (sec. 4.4) got extended by a couple of important features. The main structure and idea remained unchanged but modifications to equations and architecture were made. This again underlines the advantage of RNN to be easily extendable and to be able to integrate prior knowledge. Thus, it is also possible to adapt the network to a certain application or problem class.

Two major changes were undertaken: First, instead of using the control network to choose the new actions, only the changes of certain control parameters are determined. More important, the idea of identifying the essential Markovian state space (sec. 4.3) is further incorporated into the network by adding a bottleneck structure. Therefore, in contrast to the standard RCNN (sec. 4.4), the internal state $\bar{\mathbf{s}}_\tau$ and the approximately Markovian state $\hat{\mathbf{s}}_\tau$ now have different dimensions. Generally, one sets $\dim(\bar{\mathbf{s}}) > \dim(\hat{\mathbf{s}})$, but also the opposite direction is thinkable. However, this implies that the identity matrix between those two layers is now replaced by an additional matrix $\hat{A}$, which is also learnt during the

first step. The resulting bottleneck structure forces the network to put more emphasis on the relevant, autonomous dynamics of the RL system (eq. 1.1). It even improves the universal approximation ability (sec. 3.3) of the RCNN, as the state transition $\bar{\mathbf{s}}_{\tau+1}$ now forms a three-layered FFNN, which is itself a universal approximator (sec. 3.3.1). The required internal dimension to model the system's dynamics may therefore be reduced. By this means, also the reconstruction of the relevant Markovian state space is enforced. Besides, the improved dynamics also enhances the action selection, as it is now based on a better mapping of the dynamics. Moreover, the bottleneck structure increases the influence of the control parameters on the development of the dynamics, which supports solving the underlying credit-assignment problem. Especially short-term influences can now be better taken into account.

Analogue to equation 4.5 the optimisation task to model the dynamics (step (i)) can be represented by the following set of equations:

$$\begin{aligned} \bar{\mathbf{s}}_{\tau+1} &= f(\hat{A}\hat{\mathbf{s}}_\tau + D\mathbf{u}_\tau - \theta) \\ \bar{\mathbf{x}}_\tau &= C\bar{\mathbf{s}}_\tau \\ \text{with} \quad \hat{\mathbf{s}}_\tau &= \begin{cases} A\bar{\mathbf{s}}_\tau + B\mathbf{x}_\tau & \forall \tau \leq t \\ A\bar{\mathbf{s}}_\tau + B\bar{\mathbf{x}}_\tau & \forall \tau > t \end{cases} \end{aligned} \tag{4.8}$$

$$\sum_{t=m_-}^{T-m_+} \sum_{\tau=t-m_-}^{t+m_+} \|\bar{\mathbf{x}}_\tau - \mathbf{x}_\tau\|^2 \rightarrow \min_{A,\hat{A},B,C,D,\theta}$$

For the second step, the control network is altered as follows. Analogue to the standard RCNN (eq. 4.4) it uses the Markovian state $\hat{\mathbf{s}}_\tau$, which combines the information of the inner state $\bar{\mathbf{s}}_\tau$ and the environmental observables $\mathbf{x}_\tau$, respectively its predictions $\bar{\mathbf{x}}_\tau$, as inputs. However, it now determines the next change of control variables $\Delta\bar{\mathbf{u}}_\tau \in \mathbb{R}^K$ as an output instead of a full new action $\bar{\mathbf{u}}_\tau$. This results in the following equation:

$$\Delta\bar{\mathbf{u}}_\tau = f_u(Ff_c(E\hat{\mathbf{s}}_\tau - \mathbf{b})) \quad \forall \tau \geq t \tag{4.9}$$

where, as before, $E \in \mathbb{R}^{H\times\bar{J}}$ and $F \in \mathbb{R}^{K\times H}$, $\mathbf{b} \in \mathbb{R}^H$ is a bias, f_c an arbitrary and f_u a problem specific activation function, which can be used to scale or limit the network's action space. The calculation of $\Delta\bar{\mathbf{u}}_\tau$ instead of $\bar{\mathbf{u}}_\tau$ allows for a limited increase or decrease of the control variables and therefore avoids the learning of impossible or non-permitted changes in control. The latter is especially important with regard to gas turbine control, where the control variables can generally only be varied within a bounded interval at each time step.

As the control network now only determines the change of parameters $\Delta\bar{\mathbf{u}}_\tau$, an identity connector $\mathbf{I}_K$ of dimension K is added, which maintains the previous

control $\bar{\mathbf{u}}_{\tau-1}$ (respectively $\mathbf{u}_{\tau-1}$ for $\tau = t$). The sum of both form the new control $\bar{\mathbf{u}}_\tau$. Note that hereby the change in control can be further scaled or bounded by a fixed and problem-dependent diagonal matrix $L \in \mathbb{R}^{K\times K}$ (fig. 4.11).

Summarising, learning the optimal control (step (ii)) in the extended RCNN can be represented by the following set of equations (eq. 4.10). Anew, bold capital letters stand for fixed matrices, which are not learnt in this step.

$$
\begin{aligned}
\bar{\mathbf{s}}_{\tau+1} &= \begin{cases} f(\hat{\mathbf{A}}\hat{\mathbf{s}}_\tau + \mathbf{D}\mathbf{u}_\tau - \theta) & \forall\tau < t \\ f(\hat{\mathbf{A}}\hat{\mathbf{s}}_\tau + \mathbf{D}\bar{\mathbf{u}}_\tau - \theta) & \forall\tau \geq t \end{cases} \\
R_\tau &= \mathbf{G} f_r(\mathbf{C}\bar{\mathbf{s}}_\tau), \quad \forall\tau > t \\
\text{with} \quad \bar{\mathbf{u}}_\tau &= \bar{\mathbf{u}}_{\tau-1} + \mathbf{L} f_u(F f_c(E\hat{\mathbf{s}}_\tau - \mathbf{b})) \quad \forall\tau \geq t \\
\text{and} \quad \hat{\mathbf{s}}_\tau &= \begin{cases} \mathbf{A}\bar{\mathbf{s}}_\tau + \mathbf{B}\mathbf{x}_\tau & \forall\tau \leq t \\ \mathbf{A}\bar{\mathbf{s}}_\tau + \mathbf{B}\bar{\mathbf{x}}_\tau & \forall\tau > t \end{cases} \\
\sum_{t=m_-}^{T-m_+} \sum_{\tau>t}^{t+m_+} R_\tau &\rightarrow \max_{E,F,b}
\end{aligned}
\tag{4.10}
$$

Its architecture is depicted in figure 4.11:

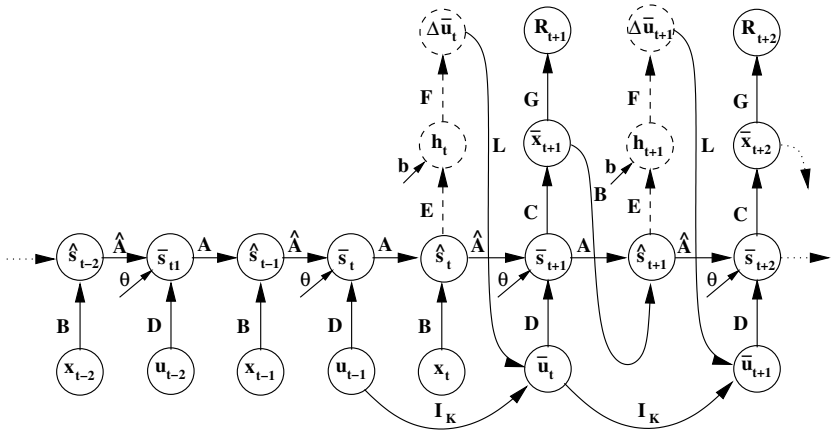


Figure 4.11: Extended recurrent control neural network architecture of step two. The control network (dashed) now determines the next change of control variable $\Delta\bar{\mathbf{u}}_\tau$ by adjusting the weight matrices E, F and the bias $\mathbf{b}$. Its sum with the previous action, which is maintained by the fixed identity matrix $\mathbf{I}_K$, forms the new action $\bar{\mathbf{u}}_\tau$ ($\tau \geq t$). Matrix L allows to further limit the change of the control parameters.

In chapter 5 the newly developed recurrent neural RL approaches are applied to a real-world problem, the control of three different gas turbine simulations.

”Any sufficiently advanced technology is indistinguishable from magic.”

Sir Arthur C. Clarke, 1917-2008, (”Profiles of The Future”, 1961 (Clarke’s third law))

CHAPTER 5

Real-World Application: Control of Gas Turbine Simulations

Gas turbines are, in contrast to burning lignite or coal, an environmental friendly way to produce electrical energy. In addition, the efficiency of gas turbines is significantly higher compared to coal-fired power plants. In combined cycle operation efficiencies of more than 60% have been achieved. Based on the short start up time and the capability of fast load changes, gas turbines are well suited to complement the fluctuating energy production from wind turbines and solar sources. They can operate on gas and oil of various qualities, while producing few emissions that can be further reduced with affordable effort. As a consequence gas turbines are, due to their good ecological properties in comparison with coal-fired power plants, increasingly deployed. Because of their high operational flexibility, they also serve as a compensation for the less predictable energy production of alternative energy sources [66].

Goals of the current developments are low emissions of NO_x, CO, and unburned hydrocarbons, while maintaining a highly efficient and smooth operation. To provide this, several parameters of gas turbines such as fuel flows, temperatures and air related settings need to be chosen optimally for the different ambient conditions and fuel properties. Furthermore, besides the instantaneous reaction of the gas turbine to parameter modifications also medium-term dynamical effects have an influence.

With the focus on controlling a gas turbine, there have already been a few attempts with feedforward neural networks [13, 14, 56, 84]. However, so far no recurrent approach has been applied. Besides, no application of reinforcement learning to gas turbine control has been reported.

5.1 Problem Description

The data was taken from industrial gas turbines as used for electrical power generation. A general schematic representation of those is depicted in figure 5.1. The amount of available real data samples ranged from sixty thousand to about one million. However, also in the case of one million samples only a selection of about seventy thousand could be used. The remaining data was recorded during static operation with very little variation of the control parameters and consequently did not provide any system development information. The time spans covered by the data sets, it was finally worked on, ranged from multiple hours, spread over a couple of days, to several months. They all contained different operating points of the turbines. Three different problem settings were examined:

(i) Combustion tuning (CT): Here the overall objective was to optimise the regarded turbine's operation for stable combustion at high load.

(ii) Emission tuning (ET): In addition to (i) also an ecological aspect, the reduction of NO_x, had to be taken into account. Moreover, further pressure intensities in the combustion chamber were considered.

(iii) Extended Emission tuning (EET): In addition to (ii) a different turbine with a more difficult dynamics was regarded. Furthermore, also a reduction of CO had to be achieved and again further pressure intensities were incorporated.

Out of the available data sets simulation models were developed as direct experiments on the gas turbines were too expensive. The simulations operate in setting (i) on 20, in (ii) on 28 and in (iii) on 40 different parameters such as electrical load (Load), exhaust temperature, temperature and pressure behind the compressor, fuel fractions, acceleration and pressure intensities in the combustion chamber (RMS). Regarded actions were in the CT setting limited modifications of pilot gas and inlet guide vane (IGV), which are two master control parameters of such turbines. In the ET setting the action space was extended to four and in the EET setting even to five different control parameters, which for both mainly consisted of the adaption of fuel fractions. Furthermore, by construction in these two settings seven and respectively four further controls were regarded in the past but assumed to be fixed in the future development of the turbine. In other words, their influence was taken into account but no changes were determined. The time grid was set to five seconds each, i.e., the simulation models were used to predict the states of the turbines five seconds ahead.

The three simulations served as a basis for testing the recurrent neural RL approaches (chap. 4). In order to guarantee the quality of the experiments first the simulations' accuracy was checked by comparing the predictions for all data

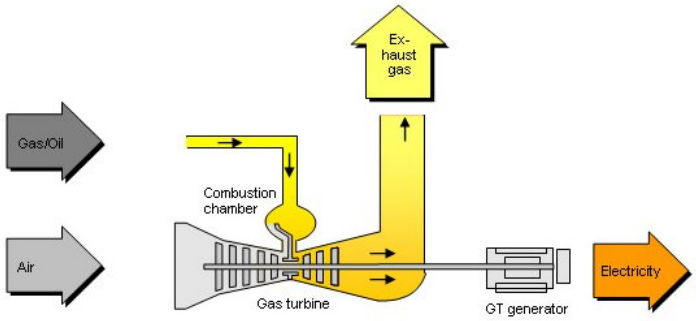


Figure 5.1: Schematic illustration of a gas turbine.

points with the true behaviour of the respective gas turbine. Furthermore, the influence of the control parameters was tested and their physical plausibility was verified. The experiments have shown that the simulations well capture the important aspects of the gas turbines' dynamics as relative one-step approximation errors of less than 3% were achieved. Moreover, the closed loop iteration over longer time periods showed a stable behaviour that kept the parameters in the valid range.

Although the problem has been already made more tractable by working on simulations, a direct application of discrete and localised RL methods, such as Q-learning or PS (sec. 2.4) on local basis functions, showed to be infeasible as the state space is still of high dimensionality, continuous, and non-Markovian. For that reason the presented recurrent neural RL approaches (chap. 4) were applied. A number of 100.000 observations each were allowed for training. With the time grid of five seconds this covers about a week of data, which is sufficient to train and update the applied methods on a regular basis. The learnt policies were tested afterwards on the respective simulation.

The reward functions used in the different settings are as follows, whereby the exact parameters $p_1, \ldots, p_n$ with $p_i \in \mathbb{R}$ $(i = 1, \ldots, n)$ are withheld due to confidentiality reasons. Here, Load $\in \mathbb{R}^+$ denotes the electrical load, RefLoad $\in \mathbb{R}^+$ a reference load of a certain data stream, $\text{RMS}_i \in \mathbb{R}^+$ with $i \in \mathbb{N}$ pressure intensities in the combustion chamber, and $\text{NO}_\text{x} \in \mathbb{R}^+$ and $\text{CO} \in \mathbb{R}^+$ the regarded emissions of the respective turbine. $\mathbf{1_A}$ stands for the indicator function of a subset A of an arbitrary set X, which is formally defined as

$$\mathbf{1_A(x)} = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases} \tag{5.1}$$

Further it is $\text{logistic}(x) = \frac{1}{1+e^{-4x}}$. The reward functions were motivated by the different problem settings. They basically resulted from the respective objective

and the considered parameters.

(i) Combustion tuning:

$$R_{\text{CT}} := p_1 \text{Load} - p_2 - p_3(\text{RMS}_1 - p_4)^2 \cdot \mathbf{1}_{(\text{RMS}_\mathbf{1} > \mathbf{p_4})}$$

Here, only one pressure intensity in the combustion chamber, RMS_1, is considered and has to be minimised to achieve an optimal reward.

(ii) Emission tuning:

$$\begin{aligned} R_{\text{ET}} \quad := \text{Load} \quad &- \quad \sum_{i=1}^{4} p_{i+4}(\text{RMS}_i - p_i)^2 \cdot \mathbf{1}_{(\text{RMS}_\mathbf{i} > \mathbf{p_i})} \\ &- \quad p_9 \cdot \text{NO}_\text{x} - p_{10} \cdot \text{logistic}\left(p_{11}(\text{NO}_\text{x} - p_{12})\right) \end{aligned}$$

In this setting already four different RMS are considered. Furthermore, the amount of NO_x has to be minimised.

(iii) Extended Emission tuning:

$$\begin{aligned} R_{\text{EET}} \quad := 1 \quad &- \quad p_9 |\text{Load} - \text{RefLoad}| \cdot \mathbf{1}_{(\text{RefLoad} > \text{Load})} \\ &- \quad \sum_{i=1}^{6} p_7 \text{logistic}\left(p_i \text{RMS}_i - p_8\right) \\ &- \quad p_{10} \tanh\left(p_{11}(\text{NO}_\text{x})^2 + p_{12}(\text{CO})^2\right) \end{aligned}$$

Here, the reward function is normalised to one. For this reason also the difference between the current Load and a reference load (RefLoad) are considered. Furthermore, six different RMS are taken into account and in addition to (ii) also CO emissions have to be minimised.

5.2 Model Description

The new developed recurrent neural RL approaches as described in chapter 4 were applied to control the different gas turbine simulations. For each, the underlying RNN was provided with the respective number of observables of the simulations as inputs and targets $\mathbf{x}_\tau$ and the number of control parameters as actions $\mathbf{u}_\tau$.

First, the extended RNN (sec. 4.3) was used within the hybrid RNN approach (sec. 4.1) to reduce the high-dimensional state spaces of the turbine simulations

and hence make the task applicable for standard table-based RL methods on a sufficiently fine-gridded discretisation. The RNN was unfolded ten time steps into past and future. It was started with a twenty dimensional state space $\hat{\mathbf{s}}_\tau$ of the RNN and incrementally applied node pruning [12, 47] to it. As described in section 4.3, the idea is to condense the state description into a lower dimensional, but approximately Markovian state, on which table-based RL algorithms can be deployed. In doing so the best compromise between generalisation performance and internal dimensionality of the state space was achieved with an only four dimensional state space. This reduced state space then served as a basis for Q-learning (RQ) (sec. 2.4.2) and prioritised sweeping (RPS) (sec. 2.4.4) to determine the optimal policy.

Second, forms of the extended RCNN (sec. 4.7) were used, as due to the problem setting, in particular the requirement to determine changes in the control parameters, the standard architecture was not applicable. The internal state dimension was set to $\bar{J} = 100$. For the combustion tuning (i) and the emission tuning (ii) setting no bottleneck structure was implemented, which means that $\dim(\hat{\mathbf{s}}) = \dim(\bar{\mathbf{s}})$ and $\hat{A} := \mathbf{I}_{\bar{J}}$. In contrast, for the extended emission tuning setting the inclusion of the essential Markovian state space reconstruction turned out to be essential. Here, the dimension of the Markovian state $\hat{\mathbf{s}}_\tau$ was set to $\hat{J} = 10$, which led to the desired bottleneck structure. The hidden state $\mathbf{h}_\tau$ of the control network was set to $H = 20$ neurons. Further, matrix L was implemented appropriately to keep the changes of the actions $\Delta\bar{\mathbf{u}}_\tau$ within the allowed limits. The chosen dimensions were effectual to generate stable results in terms of system identification and learning the optimal policy. The reward function was respectively (hard) coded into the neural architecture, which means that f_r and G were set accordingly and if needed additional clusters were implemented. As a preprocessing the parameters were scaled to the interval $[-1, 1]$. The RCNN was unfolded eight steps into the past and 24 into the future. This gave the network both, a memory length that was sufficient to identify the dynamics, and an overshooting length that enabled it to predict the consequences of its chosen actions and to calculate the desired finite sum of expected future rewards. For the same reasons as hitherto, in all networks start noise was used to handle the uncertainty of the initial state (sec. 3.5.2) and the hyperbolic tangent was implemented as activation functions f and, in the case of the RCNN, also f_c, and f_u.

5.3 Results

Table 5.1 depicts the results for the three different problem settings. Here, RefCon denotes the reference controller, whose behaviour is reflected by the original turbine data.

In all three settings the applied RCNN achieved the highest average and final reward. Also the two applied hybrid RNN approaches, RQ and RPS, showed remarkable results. In the first two settings, combustion and emission tuning, they outperformed the reference controller. Especially in the combustion tuning setting the simple RPS approach performed remarkably well. Only in the extended emission tuning setting, the hybrid RNN approach produced inferior results. Obviously, even with a reconstructed state space, the standard RL methods were not able to develop a satisfying policy. In contrast, here the extended RCNN reveals its advantage. The integrated approach allows the determination of a policy, which is superior to the reference controller. Figure 5.2 depicts the development of the achieved reward for the three different settings.

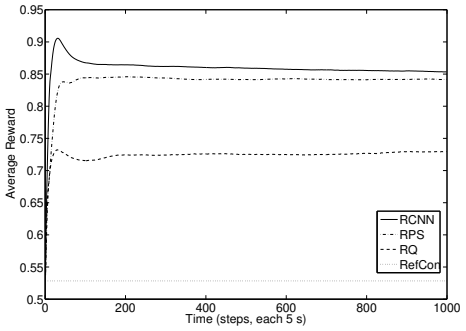
Method	Combustion tuning setting			
	average		final	
RefCon	$0.53 \pm$	0.01	$0.53 \pm$	0.01
RQ	$0.72 \pm$	0.01	$0.73 \pm$	0.01
RPS	$0.84 \pm$	0.005	$0.84 \pm$	0.005
RCNN	$\mathbf{0.86 \pm}$	$\mathbf{0.004}$	$\mathbf{0.85 \pm}$	$\mathbf{0.005}$

Method	Emission tuning setting			
	average		final	
RefCon	$0.23 \pm$	0.02	$0.23 \pm$	0.02
RQ	$0.29 \pm$	0.04	$0.29 \pm$	0.04
RPS	$0.45 \pm$	0.03	$0.51 \pm$	0.04
RCNN	$\mathbf{0.59 \pm}$	$\mathbf{0.03}$	$\mathbf{0.74 \pm}$	$\mathbf{0.03}$

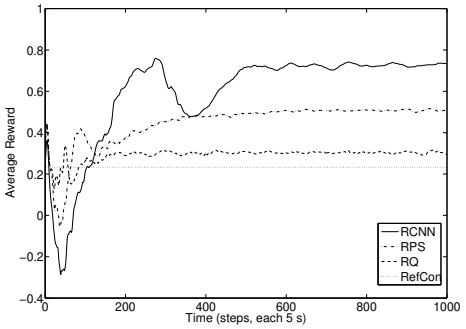
Method	Extended emission tuning setting			
	average		final	
RefCon	$0.79 \pm$	0.02	$0.79 \pm$	0.02
RQ	$0.63 \pm$	0.02	$0.57 \pm$	0.03
RPS	$0.71 \pm$	0.02	$0.68 \pm$	0.02
RCNN	$\mathbf{0.92 \pm}$	$\mathbf{0.01}$	$\mathbf{0.92 \pm}$	$\mathbf{0.02}$

Table 5.1: Average over 1000 trials and final reward after 1000 time steps achieved by the respective methods for the three regarded problem settings. Note that in each setting a different reward function was used, which implies that the results are incomparable between the three settings.

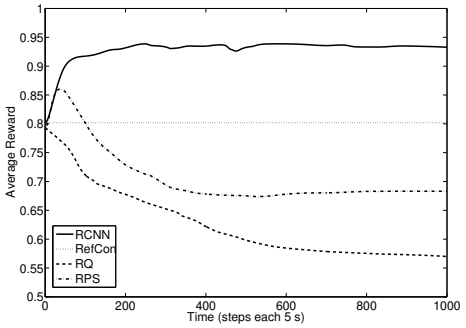
For the combustion tuning setting further the developed policies of the four regarded methods were analysed. Figure 5.3 illustrates their performance behaviours after reaching a stable operating point. Figure 5.3(a) compares the respective mean setting of the control parameters, pilot gas and IGV. It shows that



(a) Combustion tuning setting



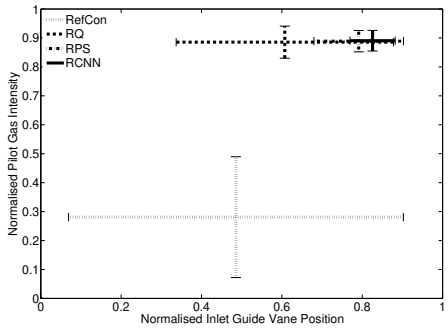
(b) Emission tuning setting



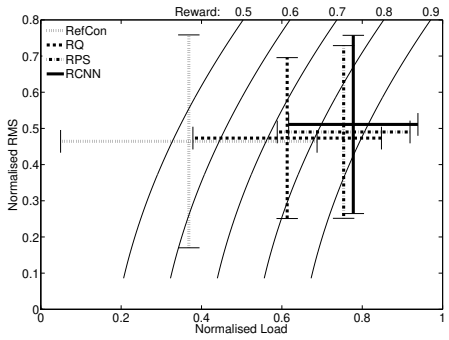
(c) Extended emission tuning setting

Figure 5.2: Evaluation of the reward development for the different controllers for (a) the combustion tuning and (b) the emission tuning setting. The reward is plotted for the reference controller (RefCon), RNN based Q-learning (RQ), RNN based prioritised sweeping (RPS), and the extended RCNN averaged over 1000 trials with different starting points.

the presented recurrent neural RL controllers developed the most stable policies as their standard deviation is low. Notably, these four methods approach very similar operating points. Figure 5.3(b) depicts the two major performance indicators, Load and RMS. It confirms the good results of the novel methods as they reach the highest Load level with only a minor increase in RMS. To ease interpretation "iso-reward" curves are plotted, which indicate identical reward for different combinations of the performance indicators.



(a) Control Parameters pilot gas and IGV



(b) Performance Indicators Load and RMS

Figure 5.3: Comparison of the final operation points reached by different controllers in the combustion tuning setting. The plots show the mean value and standard deviation in (a) the space of the control parameters and (b) the space of the performance indicators. In the latter "iso-reward" curves indicate identical reward in this space.

”Science, never solves a problem without creating ten more.”

Georg Bernhard Shaw, 1856-1950

CHAPTER 6

Conclusion

In this thesis a novel connection between reinforcement learning and recurrent neural networks is presented. Its practicality to solve high-dimensional and partially observable RL problems with continuous state and action spaces data-efficiently has been shown on several benchmarks and an application to gas turbine simulations. As a preposition theoretical results on RNN have been developed. The thesis therefore contributes valuable results for both research fields, reinforcement learning and recurrent neural networks.

In summary the following contributions have been made:

(i) A proof for the universal approximation ability of RNN: It has been proven that RNN in state space model form can approximate any open dynamical system with an arbitrary accuracy. The result is of great importance as it forms a theoretical basis for recurrent neural network research in general, but also for the application of RNN to reinforcement learning.

(ii) A demonstration of RNN's ability to learn long-term dependencies: It has been shown that RNN unfolded in time and trained with a shared weight extension of the backpropagation algorithm are, in opposition to an often stated opinion, well able to learn long-term dependencies. Using shared weights in combination with a reasonable learning algorithm like pattern-by-pattern learning and a proper weight initialisation the problem of a vanishing gradient becomes a minor issue. Due to shared weights RNN even possess an internal regularisation mechanism, which keeps the error flow up and allows for an information transport over at least a hundred time steps. The analysis confirms that RNN are valuable in system identification and forecasting. However, it is of particular importance for reinforcement learning where a long horizon is necessary to evaluate the policies.

(iii) Practical details on an improved model-building with RNN: A couple of approaches in RNN model-building were presented, which have shown to be very useful in practical applications. Hereby, methods to deal with uncertainty in the data as well as in the initial state of the finitely unfolded RNN were described. Additionally, ways for an optimal weight initialisation were discussed. These aspects are of importance for an optimal system identification and also positively influenced the performance of RNN applied to RL problems.

(iv) An RNN based method for an optimal state-space reconstruction as well as minimisation: A novel RNN based method to reconstruct or minimise the state space of a partially observable system was introduced. RNN were used to reconstruct or minimise a system's state space within their inner state. In doing so a problem can be made more tractable. The method served as a basis for recurrent neural reinforcement learning. Still, it is itself valuable as it allows a kind of feature selection, where in contrast to classical approaches, the features provide across-the-time information.

(v) A hybrid RNN approach to solve RL problems with a combination of RNN and standard RL algorithms: The hybrid RNN approach represents a combination of state space reconstruction and respectively minimisation with RNN (iv) and RL. It is a two step method. In the first step the RNN is used to create a suitable state space, which standard RL methods can deal with. In the second step the internal state of the RNN then serves as a basis for those RL methods, which are applied to learn the optimal policy. The approach is quite simple, but has shown remarkable results. For the first time the cart-pole problem could be solved with only one observable variable.

(vi) An RNN to explicitly model POMDP: A special RNN architecture was developed for the mapping of POMDP and the reconstruction of a (minimal) approximately Markovian state space. Its structure is adapted to POMDP. So it considers actions as separate inputs and allows to explicitly reconstruct the RL system's state out of the observables and the networks inner state. The network can be used to enhance the hybrid RNN approach (v). This has shown to be particularly of advantage for RL algorithms, which are by construction unable to deal with partially observable environments. Their application range can be enlarged through a preprocessing of the RL system's state space with the described RNN.

(vii) The recurrent control neural network in its standard and its extended form: The RCNN forms the desired connection between RL and RNN. It combines system identification and learning of the policy of an RL problem within one integrated recurrent neural network. The approach is model-based and by construction able to solve high-dimensional and partially observable RL problems with continuous state and action spaces in a data-efficient manner. On this note it also offers a good approach to break Bellman's curse of dimensionality [7]. In contrast to most RL methods it makes no use of a value function. Its learning of a policy can rather be referred to as virtual Monte-Carlo policy gradient method.
The application to the classical cart-pole problem demonstrated the capabilities of the RCNN, especially in terms of data-efficiency and robustness. The results on the mountain car problem further showed that the RCNN is able to learn a potentially optimal policy.
In an extended version the RCNN further incorporates the idea of a minimal Markovian state space reconstruction and allows for the calculation of changes in the control parameters instead of absolute values only. This has shown to be a clear advantage for real-world applications.

(viii) An application to gas turbine control: The application on gas-turbine control was motivated by a research project at Siemens AG. Within the limits of confidentiality the problem setting was described and its particular difficulties were pointed out. For a solution the novel recurrent neural RL methods were applied. Overall, the results were remarkable as in comparison to the given reference controller a substantial performance gain could be achieved. Furthermore, the methods have been shown to significantly improve the turbines stability and its lifetime by guaranteeing the accustomed high performance. However, in the application to the three different simulations of gas turbines it became also evident that the integrated RCNN approach is more powerful than the hybrid one. Nevertheless, despite its simplicity the latter showed a remarkable performance.

Future research can be done in several directions. On the one hand the theory of RNN can be further developed. Thereby proofs for numerous assumptions, like the details on improved model-building, would be desirable. Besides, a comparison between the multiple recurrent neural network types might be of interest. On the other hand one might further enhance the novel neural RL approaches. Here a combination with the theory on sparse matrices, which have shown very good results on long-term learning [78], large neural networks [106] or also safe exploration [31] would be thinkable. In addition, an application to real gas turbines and similar real-world applications is aspired. Coming along with that is a constant improvement and adaptation of the networks to each respective problem setting.

Bibliography

[1] M. Abramson, P. Pachowicz, H. Wechsler, Competitive reinforcement learning in continuous control tasks, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN), Portland, OR, 2003.

[2] B. Bakker, Reinforcement learning with long short-term memory, in: T. G. Dietterich, S. Becker, Z. Ghahramani (eds.), Advances in Neural Information Processing Systems, No. 14, MIT Press, Cambridge, MA, pp. 1475–1482, 2002.

[3] B. Bakker, The state of mind: Reinforcement learning with recurrent neural networks, Ph.D. thesis, Leiden University, 2004.

[4] B. Bakker, Reinforcement learning by backpropagation through an LSTM model/critic, in: Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL), Honolulu, HI, pp. 127–134, 2007.

[5] A. Barto, R. S. Sutton, C. Anderson, Neuron-like adaptive elements that can solve difficult learning control problems, IEEE Transactions on Systems, Man, and Cybernetics (13), pp. 834–846, 1983.

[6] R. E. Bellman, Dynamic Programming and Stochastic Control Processes, Rand Corporation, 1957.

[7] R. E. Bellman, Adaptive Control Processes: A Guided Tour, Princeton University Press, Princton, NJ, 1961.

[8] Y. Bengio, P. Simard, P. Frasconi, Learning long-term dependencies with gradient descent is difficult, IEEE Transactions on Neural Networks 5 (2), pp. 157–166, 1994.

[9] D. P. Bertsekas, Dynamic Programming: Deterministic and Stochastic Models, Prentice Hall, Englewood Cliffs, NJ, 1987.

[10] D. P. Bertsekas, Dynamic Programming and Optimal Control, vol. 1-2, Athena Scientific, Belmont, MA, 1995.

[11] D. P. Bertsekas, J. N. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, Belmont, MA, 1996.

[12] C. M. Bishop, Neural Networks for Pattern Recognition, Clarendon Press, Oxford, 1995.

[13] N. W. Chbat, et.al., Estimating gas turbine internal cycle parameters using a neural network, International Gas Turbine and Aeroegine Congress and Exhibition, 1996.

[14] N. Chiras, C. Evans, D. Rees, Nonlinear gas turbine modelling using feedforward neural networks, in: Proceedings of ASME Turbo Expo Congress, Amsterdam, 2002.

[15] G. Cybenko, Approximation by superpositions of a sigmoidal function, in: Mathematics of Control, Signals and Systems, Springer, New York, pp. 303–314, 1989.

[16] K.-L. Du, M. N. S. Swamy, Neural Networks in a softcomputing framework, Springer, London, 2006.

[17] J. L. Elman, Finding structure in time, Cognitive Science 14, pp. 179–211, 1990.

[18] E. A. Feinberg, A. Shwartz, Handbook of Markov Decision Processes, Kluwer, 2002.

[19] K. I. Funahashi, On the approximate realization of continuous mappings by neural networks, Neural Networks 2, pp. 183–192, 1989.

[20] A. R. Gallant, H. White, There exists a neural network that does not make avoidable mistables, in: Proceedings of the Second Annual IEEE Conference on Neural Networks, vol. 1, IEEE Press, pp. 657–664, 1988.

[21] H. Gatignon, Statistical Analysis of Management Data, Springer US, 2003.

[22] F. Gers, N. Schraudolph, J. Schmidhuber, Learning precise timing with LSTM recurrent networks, Journal of Machine Learning Research 3, pp. 115–143, 2002.

[23] C. L. Giles, C. B. Miller, D. Chen, G. Z. Sun, H. H. Chen, Y. C. Lee, Extracting and learning an unknown grammar with recurrent neural networks, in: J. E. Moody, S. J. Hanson, R. P. Lippmann (eds.), Advances in Neural Information Processing Systems, vol. 4, Morgan Kaufmann Publishers, pp. 317–324, 1992.

[24] P. E. Gill, W. Murray, M. H. Wright, Practical Optimization, Academic Press, London, 1981.

[25] F. Gomez, Robust non-linear control through neuroevolution, Ph.D. thesis, University of Texas, Austin, 2003.

[26] F. Gomez, R. Miikkulainen, 2-D balancing with recurrent evolutionary networks, in: Proceedings of the International Conference on Artificial Neural Networks (ICANN-98), Springer, Skovde, pp. 425–430, 1998.

[27] M. Gopal, Modern Control System Theory, Wiles Eastern Limited, 1984.

[28] R. Grothmann, Multi-agent market modeling based on neural networks, Ph.D. thesis, University of Bremen, Bremen, 2002.

[29] R. Grothmann, A. M. Schaefer, S. Clarke, A. Zaeh, Telepresent production: Overcoming the network problem with error correction neural networks, in: Proceedings of the International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV), Munich, 2005.

[30] B. Hammer, On the approximation capability of recurrent neural networks, in: International Symposium on Neural Computation, 1998.

[31] A. Hans, D. Schneegass, A. M. Schaefer, S. Udluft, Safe exploration for reinforcement learning, in: Proceedings of the European Symposium on Artificial Neural Networks (ESANN), Bruges, 2008.

[32] M. Harmon, S. Harmon, Reinforcement learning: A tutorial, 1996.

[33] S. Haykin, Neural Networks: A Comprehensive Foundation, Macmillan, New York, 1994.

[34] S. Haykin, J. Principe, T. Sejnowski, J. McWhirter, New Directions in Statistical Signal Processing: From Systems to Brain, MIT Press, Cambridge, MA, 2006.

[35] S. Hochreiter, The vanishing gradient problem during learning recurrent neural nets and problem solutions, International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 6 (2), pp. 107–116, 1998.

[36] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, Gradient flow in recurrent nets: The difficulty of learning long-term dependencies, in: J. F. Kolen, S. Kremer (eds.), A Field Guide to Dynamical Recurrent Networks, IEEE Press, pp. 237–243, 2001.

[37] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Computation 9 (8), pp. 1735–1780, 1997.

[38] K. Hornik, M. Stinchcombe, H. White, Multi-layer feedforward networks are universal approximators, Neural Networks 2, pp. 359–366, 1989.

[39] H. Jaeger, Adaptive nonlinear system identification with echo state networks, in: S. Becker, S. Thrun, K. Obermayer (eds.), Advances in Neural Information Processing Systems, vol. 15, MIT Press, Cambridge, pp. 593–600, 2003.

[40] H. Jaeger, W. Maass, J. Principe, Special issue on echo state networks and liquid state machines, vol. 20 (3) of Neural Networks, 2007.

[41] L. P. Kaelbling, M. L. Littman, A. R. Cassandra, Planning and acting in partially observable stochastic domains, Journal of Artificial Intelligence 101 (1-2), pp. 99–134, 1998.

[42] L. P. Kaelbling, M. L. Littman, A. P. Moore, Reinforcement learning: A survey, Journal of Artificial Intelligence 4, pp. 237–285, 1994.

[43] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, Science 220 (4598), pp. 671–680, 1983.

[44] J. F. Kolen, S. C. Kremer, A Field Guide to Dynamical Recurrent Networks, IEEE Press, 2001.

[45] V. R. Konda, J. N. Tsitsikilis, On actor-critic algorithms, SIAM Journal on Control and Optimization 42(4), pp. 1143–1166, 2003.

[46] M. G. Lagoudakis, R. Parr, Least-squares policy iteration, Journal of Machine Learning Research, pp. 1107–1149, 2003.

[47] Y. LeCun, J. S. Denker, S. A. Solla, Optimal brain damage, in: D. Touretzky (ed.), Advances in Neural Information Processing Systems 2, pp. 598–605, 1990.

[48] D. G. Luenberger, Introduction to linear and nonlinear Programming, Addison-Wesley, Reading, 1973.

[49] D. P. Mandic, J. A. Chambers, Recurrent neural networks for prediction: Learning algorithms, architectures and stability, in: S. Haykin (ed.), Adaptive and Learning Systems for Signal Processing, Communications and Control, John Wiley & Sons, Chichester, 2001.

[50] A. A. Markov, Markov Chains, chap. Appendix B: Extension of the limit theorems of probability theory to a sum of variables connected in a chain, John Wiley & Sons, reprint 1971.

[51] L. R. Medsker, L. C. Jain, Recurrent neural networks: Design and application, vol. 1 of comp. intelligence, CRC Press international, 1999.

[52] N. Metropolis, S. Ulam, The Monte Carlo method, Journal of the American Statistical Association 44, pp. 335–341, 1949.

[53] N. Meuleau, L. Peshkin, K. Kee-Eung, L. P. Kaebling, Learning finite-state controllers for partially observable environments, in: Proceedings of the Fifteenth International Conference on Uncertainty in Artificial Intelligence (UAI-99), Morgan Kaufmann, San Francisco, CA, pp. 427-436, 1999.

[54] M. Minsky, Steps towards artificial intelligence, in: Proceedings of the Institute of Radio Engineers, pp. 8–30, 1961.

[55] A. W. Moore, C. G. Atkeson, Prioritized sweeping: Reinforcement learning with less data and less time, Machine Learning 13, pp. 103–130, 1993.

[56] I. T. Nabney, D. C. Cressy, Neural network control of a gas turbine, Neural Computing and Applications 4 (4), pp. 198–208, 1996.

[57] R. Neuneier, H. G. Zimmermann, How to train neural networks, in: G. B. Orr, K.-R. Mueller (eds.), Neural Networks: Tricks of the Trade, Springer Verlag, Berlin, pp. 373–423, 1998.

[58] B. Pearlmutter, Gradient calculations for dynamic recurrent neural networks: A survey, IEEE Transactions on Neural Networks 6 (5), pp. 1212–1228, 1995.

[59] J. Peters, E. Theodorou, S. Schaal, Policy gradient methods for machine learning, in: Proceedings of INFORMS Conference of the Applied Probability Society, 2007.

[60] J. Peters, S. Vijayakumar, S. Schaal, Natural actor-critic, in: J. Gama, et al. (eds.), Machine Learning: ECML 2005, No. 3720 in Lecture Notes in Artificial Intelligence, Springer, pp. 280–291, 2005.

[61] D. Precup, R. Sutton, S. Singh, Theoretical results on reinforcement learning with temporally abstract behaviors, in: Proceedings of the 10th European Conference on Machine Learning (ECML), pp. 382–393, 1998.

[62] D. Prokhorov, Toward effective combination of off-line and on-line training in ADP framework, in: Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL), Honolulu, HI, pp. 268–271, 2007.

[63] D. Prokhorov, Toyota Prius HEV neurocontrol, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN), MIT Press, Orlando, pp. 2129–2134, 2007.

[64] D. Prokhorov, D. C. Wunsch, Adaptive critic designs, IEEE Transactions on Neural Networks 8 (5), pp. 997–1007, 1997.

[65] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, Wiles Series in Probability and Statistics, John Wiley & Sons, 2005.

[66] P. Ratliff, P. Garbett, W. Fischer, The new Siemens gas turbine SGT5-8000H for more costumer benefit, VGB PowerTech, 2007.

[67] M. Riedmiller, Supervised learning in multilayer perceptrons - from backpropagation to adaptive learning techniques, International Journal of Computer Standards and Interfaces 16 (5), pp. 265–278, 1994.

[68] M. Riedmiller, Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method, in: J. Gama, et al. (eds.), Machine Learning: ECML 2005, No. 3720 in Lecture Notes in Artificial Intelligence, Springer, pp. 317–328, 2005.

[69] M. Riedmiller, H. Braun, A direct adaptive method for faster backpropagation learning: The Rprop algorithm, in: Proceedings of the IEEE International Conference on Neural Networks, IEE Press, pp. 586–591, 1993.

[70] M. Riedmiller, J. Peters, S. Schaal, Evaluation of policy gradient methods and variants on the cart-pole benchmark, in: Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL), Honolulu, HI, pp. 254–261, 2007.

[71] D. E. Rumelhart, G. E. Hinton, R. J. Williams, Learning internal representations by error propagation, in: D. E. Rumelhart, J. L. McClelland, et al. (eds.), Parallel Distributed Processing: Explorations in The Microstructure of Cognition, vol. 1, MIT Press, Cambridge, MA, pp. 318–362, 1986.

[72] A. L. Samuel, Some studies in machine learning using the game checkers, IBM Journal on Research and Developement (3), pp. 210–229, 1959.

[73] A. M. Schaefer, D. Schneegass, V. Sterzing, S. Udluft, A neural reinforcement learning approach to gas turbine control, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN), MIT Press, Orlando, pp. 1691-1696, 2007.

[74] A. M. Schaefer, D. Schneegass, V. Sterzing, S. Udluft, Data-efficient recurrent neural reinforcement learning for gas turbine control, Tech. rep., Siemens AG, 2008.

[75] A. M. Schaefer, S. Udluft, Solving partially observable reinforcement learning problems with recurrent neural networks, in: A. Nowe, et al. (eds.), Reinforcement Learning in Non-Stationary Environments, Workshop Proceedings of the European Conference on Machine Learning (ECML), pp. 71-81, 2005.

[76] A. M. Schaefer, S. Udluft, H. G. Zimmermann, The recurrent control neural network, in: Proceedings of the European Symposium on Artificial Neural Networks (ESANN), Bruges, pp. 319–324, 2007.

[77] A. M. Schaefer, S. Udluft, H. G. Zimmermann, A recurrent control neural network for data efficient reinforcement learning, in: Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL), Honolulu, HI, pp. 151–157, 2007.

[78] A. M. Schaefer, S. Udluft, H. G. Zimmermann, Learning long term dependencies with recurrent neural networks, NeuroComputing 71 (13-15), pp. 2481–2488, 2008.

[79] A. M. Schaefer, H. G. Zimmermann, Recurrent neural networks are universal approximators, International Journal of Neural Systems 17 (4), pp. 253–263, 2007.

[80] J. Schmidhuber, Reinforcement learning in markovian and non-markovian environments, in: D. S. Lippman, J. E. Moody, D. S. Touretzky (eds.), Advances in Neural Information Processing Systems, vol. 3, Morgan Kaufmann, San Mateo, CA, pp. 500–506, 1991.

[81] J. Schmidhuber, F. Gers, D. Eck, Learning nonregular languages: A comparison of simple recurrent networks and LSTM, Neural Computation 14 (9), pp. 2039–2041, 2002.

[82] D. Schneegass, S. Udluft, T. Martinetz, Neural rewards regression for near-optimal policy identification in markovian and partial observable environments, in: M. Verleysen (ed.), Proceedings of the European Symposium on Artificial Neural Networks (ESANN), pp. 301–306, 2007.

[83] N. Schraudolph, Rapid stochastic gradient descent: Accelerating machine learning, slides of a lecture on stochastic meta-descent, 2007.

[84] Q. Song, et. al., An integrated robust/neural controller with gas turbine applications, in: IEEE Conference on Control Applications, pp. 411-415, 1994.

[85] A. Soofi, L. Cao, Modeling and Forecasting Financial Data, Techniques of Nonlinear Dynamics, Kluwer Academic Publishers, 2002.

[86] M. H. Stone, The generalized weierstrass approximation theorem, Mathematics Magazine 21, 1948.

[87] M. J. A. Strens, A. W. Moore, Direct policy search using paired statistical tests, in: Proceedings of the Eighteenth International Conference on Machine Learning (ICML), Williams College, MA, 2001.

[88] R. S. Sutton, Learning to predict by the methods of temporal differences, Machine Learning 3, pp. 9–44, 1988.

[89] R. S. Sutton, A. Barto, Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning), MIT Press, Cambridge, MA, 1998.

[90] R. S. Sutton, D. McAllester, S. Singh, Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, in: Advances in Neural Information Processing Systems 12, 2000.

[91] G. J. Tesauro, TD-gammon, a self-teaching backgammon program, achieves master-level play, Neural Computation 6(2), pp. 215–219, 1994.

[92] J. N. Tsitsikilis, B. Van Roy, An analysis of temporal difference learning with function approximation, IEEE Transactions on Automatic Control 42(5), pp. 674–690, 1997.

[93] C. Watkins, Learning from delayed rewards, Ph.D. thesis, University of Cambridge, 1989.

[94] K. Weierstrass, Über die analytische Darstellbarkeit sogenannter willkürlicher Functionen einer reellen Veränderlichen, Tech. Rep. II, Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften zu Berlin, 1885.

[95] P. J. Werbos, Beyond regression: New tools for prediction and analysis in the behavioral sciences, Ph.D. thesis, Harvard University, 1974.

[96] P. J. Werbos, Neural networks & the human mind: New mathematics fits humanistic insight, in: Proceedings of IEEE International Conference on Systems, Man and Cybernetics, IEEE, Chicago, 1992.

[97] P. J. Werbos, The Roots of Backpropagation. From Ordered Derivatives to Neural Networks and Political Forecasting, John Wiley & Sons, New York, 1994.

[98] D. A. White, D. A. Sofge, Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches, Van Nostrand Reinhold, New York, 1992.

[99] A. Wieland, Evolving neural network controllers for unstable systems, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN), vol. II, Piscataway, NJ, pp. 667-673, 1991.

[100] R. J. Williams, L. C. Baird, Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems, Tech. Rep. NU-CCS-93-11, Northeastern Universty, College of Computer Science, Boston, MA, 1993.

[101] R. J. Williams, D. Zipser, A learning algorithm for continually running fully recurrent neural networks, Neural Computation, 1989.

[102] R. J. Williams, D. Zipser, Gradient-based learning algorithms for recurrent connectionist networks, in: Y. Chauvin, D. E. Rumelhart (eds.), Backpropagation: Theory, Architectures, and Applications, Erlbaum, Hillsdale, NJ, 1990.

[103] H. G. Zimmermann, Neuronale Netze als Entscheidungskalkül, in: H. Rehkugler, H. G. Zimmermann (eds.), Neuronale Netze in der Ökonomie: Grundlagen und ihre finanzwirtschafliche Anwendung, Vahlen, Munich, pp. 1–88, 1994.

[104] H. G. Zimmermann, L. Bertolini, R. Grothmann, A. M. Schaefer, C. Tietz, A technical trading indicator based on dynamical consistent neural networks, in: Proceedings of the International Conference on Artificial Neural Networks (ICANN), vol. 2, Springer, Athens, pp. 654-663, 2006.

[105] H. G. Zimmermann, R. Grothmann, A. M. Schaefer, C. Tietz, Dynamical consistent recurrent neural networks, in: D. Prokhorov (ed.), Proceedings of the International Joint Conference on Neural Networks (IJCNN), MIT Press, Montreal, pp. 1537–1541, 2005.

[106] H. G. Zimmermann, R. Grothmann, A. M. Schaefer, C. Tietz, Identification and forecasting of large dynamical systems by dynamical consistent neural networks, in: S. Haykin, J. Principe, T. Sejnowski, J. McWhirter (eds.), New Directions in Statistical Signal Processing: From Systems to Brain, MIT Press, pp. 203–242, 2006.

[107] H. G. Zimmermann, R. Grothmann, A. M. Schaefer, C. Tietz, Energy future price forecasting by dynamical consistent neural networks, working paper, Siemens AG, Munich, 2007.

[108] H. G. Zimmermann, R. Neuneier, The observer-observation dilemma in neuro-forecasting, Advances in Neural Information Processing Systems 10, pp. 179–206, 1998.

[109] H. G. Zimmermann, R. Neuneier, Neural network architectures for the modeling of dynamical systems, in: J. F. Kolen, S. Kremer (eds.), A Field Guide to Dynamical Recurrent Networks, IEEE Press, pp. 311–350, 2001.

[110] H. G. Zimmermann, R. Neuneier, R. Grothmann, An approach of multi-agent FX-market modeling based on cognitive systems, in: Proceedings of the International Conference on Artificial Neural Networks (ICANN), Springer, pp. 767–774, 2001.

[111] H. G. Zimmermann, R. Neuneier, R. Grothmann, Modeling of dynamical systems by error correction neural networks, in: A. Soofi, L. Cao (eds.), Modeling and Forecasting Financial Data, Techniques of Nonlinear Dynamics, Kluwer Academic Publishers, pp. 237–263, 2002.