Technische Universität München

Fakultät für Informatik
Lehrstuhl VI – Echtzeitsysteme und Robotik

REINFORCEMENT LEARNING IN SUPERVISED PROBLEM DOMAINS

Thomas F. Rückstieß

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Daniel Cremers
Prüfer der Dissertation	1.	Univ.-Prof. Dr. Patrick van der Smagt
	2.	Univ.-Prof. Dr. Hans Jürgen Schmidhuber

Die Dissertation wurde am 30. 06. 2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18. 09. 2015 angenommen.

Thomas Rückstieß: *Reinforcement Learning in Supervised Problem Domains* © 2015

EMAIL:
thomas@rueckstiess.net

ABSTRACT

Despite continuous advances in computing technology, today's brute force data processing approaches may not provide the necessary advantage to win the race against the ever-growing amount of data that can be witnessed over the last decades. In this thesis, we discuss novel methods and algorithms that are capable of directing attention to relevant details and analysing it in sequence to overcome the processing bottleneck and to keep up with this data explosion.

In the first of three parts, a novel exploration technique for Policy Gradient Reinforcement Learning is presented which replaces traditional additive random exploration with state-dependent exploration, exploring on a higher, more strategic level. We will show how this new exploration method converges faster and finds better global solutions than random exploration can.

The second part of this thesis will introduce the concept of "data consumption" and discuss means to minimise it in supervised learning tasks by deriving classification as a sequential decision process and making it accessible to Reinforcement Learning methods. Depending on previously selected features and the internal belief state of a classifier a next feature is chosen by a sequential online feature selection that learns which features are most informative at each given time step. In experiments this attentive hybrid learning system shows significant reduction in required data for correct classification.

Finally, the third major contribution of this thesis is a novel sequence learning approach that learns an explicit contextual state while traversing a sequence. This context helps distinguish the current input and mitigates the need for a predictor capable of dealing with sequential data. We show the close relationship to concepts from theoretical computer science, in particular that of deterministic finite automata and regular languages and demonstrate experimentally the capabilities of this hybrid algorithm.

All three parts share in common a tight integration of Reinforcement Learning and Supervised Learning which not only delivers an orthogonal view onto this research but also establishes for the first time a general framework of such hybrid algorithms.

ZUSAMMENFASSUNG

Trotz des ständigen technischen Fortschritts in der Computertechnologie ist es durchaus möglich, dass die heutigen Holzhammer-Methoden der Datenanalyse uns nicht den nötigen Vorteil bringen, um das Rennen gegen das stetige Datenwachstum zu gewinnen, das in den letzten Jahrzehnten zu beobachten ist. In dieser Dissertation werden neue Methoden und Algorithmen diskutiert, die in der Lage sind, ihre Aufmerksamkeit auf die relevanten Details zu richten und diese sequentiell zu verarbeiten, um den Flaschenhals der Informationsverarbeitung zu überwinden und mit der Datenexplosion Schritt zu halten.

Im ersten von drei Teilen wird eine neuartige Explorationsmethode für Reinforcement Learning (bestärkendes Lernen) mit Policy Gradients vorgestellt, welches die traditionelle Art des additiven Explorierens durch zustandsabhängige Exploration ersetzt und auf einem höheren und mehr strategischen Level arbeitet. Wir zeigen, dass diese neue Form der Exploration schneller konvergiert und bessere und globalere Lösungen finden kann als zufällige Exploration.

Der zweite Teil dieser Arbeit stellt das Konzept des "Datenkonsums" vor und diskutiert Mittel, um diesen für überwachte Lernvorgänge zu minimieren. Dies wird ermöglicht indem Klassifizierung als sequenzieller Entscheidungsprozess hergeleitet und dadurch bestärkenden Lernmethoden zugänglich gemacht wird. Abhängig von vorherig ausgewählten Features und des internen Zustands eines Klassifizierers wählt eine sequenzielle Komponente ein neues Feature in Echtzeit aus, indem es lernt, welches Feature den höchsten Informationsgehalt zum jeweiligen Zeitpunkt trägt. In Experimenten zeigt dieses hybride Lernsystem eine signifikante Verringerung der nötigen Datenmenge für die korrekte Klassifizerung.

Der dritte bedeutende Beitrag dieser Dissertation beschreibt eine neue sequenzielle Lernmethode, die während des Traversierens einer Sequenz einen expliziten kontextuellen Zustand aufbaut. Dieser Kontext unterstützt die Charakterisierung der aktuellen Eingabe und ermöglicht das Auseinanderhalten zeitlicher Eingabesignale ohne die Hilfe einer sequentiellen Klassifikationsmethode. Wir zeigen die enge Verwandtschaft zu Konzepten aus der Theoretischen Informatik auf, insbesondere zur Automatentheorie und Regulären Sprachen, und demonstrieren die Möglichkeiten dieses hybriden Algorithmus experimentell.

Alle drei Teile haben eine enge Integration von bestärkenden und überwachten Lernmethoden gemeinsam. Dies stellt nicht nur eine alternative Ansicht auf die hier vorgestellten Forschungsergebnisse dar, sondern etabliert zum ersten Mal ein allgemeines Rahmensystem solcher kombinierter Algorithmen.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Prof. Jürgen Schmidhuber and Prof. Patrick van der Smagt for their help and support. Without either of them, this thesis would not have been possible. I thank Jürgen for believing in me and letting me join his group, for his inspiring vision and valuable conversations, and for his support and funding during the first years of my Ph.D. I want to thank Patrick for jumping in and taking over the supervision without knowing me much, for funding my work in the later years of my Ph.D., and for the interesting lunch discussions during my visits. His constructive and criticial suggestions and diligent proofreading have helped improve this thesis tremendously.

Special thanks go to my colleagues at TUM (a.k.a. "die Buam"), for the many great and interesting years together. Christian Osendorfer, my "roommate" for many years, was always there to bounce off ideas, discuss crazy theories and help out with complex mathematical problems. Thanks go to Frank Sehnke, Alex Graves and Martin Felder, the other Buam in the team. We worked on some great projects together, always had each other's backs and made the years of Ph.D. slavery a fun experience. The other contributors of PyBrain, in particular Justin Bayer, Tom Schaul and Daan Wierstra, also deserve a mention for their great work on this joint project. Everyone at the chair of Robotics and Embedded Systems, especially Prof. Alois Knoll, Dr. Gerhard Schrott, our secretaries Monika Knürr, Gisela Hibsch and Amy Bücherl, have helped and supported me in various ways over the years and made my time at TUM more about the research and less about bureaucracy.

A big Dankeschön goes to all my friends. I'm lucky to have met so many great people, both during my time in Tübingen, and after my move to München. They supported me in every decision, were there for me during good and bad times, and kept me sane and happy over all these years. I miss you all incredibly.

I also want to thank my parents Wolfgang and Berta for a happy and unburdened upbringing, for giving me the opportunity to make my own choices in life and for always giving me good advise for the important decisions.

Finally, I would like to thank my wonderful partner Steve Pennells for supporting me unconditionally during all this time, for his sacrifices and his endless patience over the last years and for being such a great companion on our journey around the globe and through life.

CONTENTS

ABSTRACT iii

ZUSAMMENFASSUNG iv

ACKNOWLEDGEMENTS vi

I Overview 1

1 INTRODUCTION 3

2 OUTLINE 5

3 CONTRIBUTIONS 7
- 3.1 Scientific Contributions 7
- 3.2 Technical Contributions 8

II Literature Review 11

4 FUNCTION APPROXIMATION 13
- 4.1 General 13
- 4.2 Function Approximation in Reinforcement Learning . . . 14
- 4.3 Linear Regression 15
- 4.4 Logistic and Multinomial Regression 17
- 4.5 Neural Networks 18
  - 4.5.1 Feed-Forward Neural Networks 19
  - 4.5.2 Recurrent Neural Networks and LSTMs 22
- 4.6 Locally Weighted Projection Regression 24
- 4.7 Sequence Learning 24

5 REINFORCEMENT LEARNING 27
- 5.1 General Remarks and Notation 27
  - 5.1.1 Formal Definition of Reinforcement Learning . . . 28
- 5.2 Model-based Reinforcement Learning 31
- 5.3 Value-Based Reinforcement Learning 31
  - 5.3.1 Discrete Value-Based RL 32
  - 5.3.2 Continuous Value-Based RL 35
- 5.4 Direct Reinforcement Learning 41
  - 5.4.1 Overall performance measure 41

5.4.2 Finite Difference Methods 41
5.4.3 Likelihood Ratio Methods 42
5.5 Exploration 45
5.5.1 Exploration in Discrete Action Spaces 45
5.5.2 Exploration in Continuous Action Spaces 46

III Methods and Experiments 49

6 STATE-DEPENDENT EXPLORATION 51
6.1 Introduction 51
6.2 Exploration in Parameter Space 53
6.3 State-Dependent Exploration 55
6.3.1 REINFORCE for General Multi-Dimensional FA . . 56
6.3.2 Derivation from REINFORCE to SDE 58
6.3.3 Adaptive Exploration Variance 61
6.3.4 Stochastic Policies 62
6.3.5 Negative Variances 62
6.3.6 State-Dependent Exploration for Value-Based RL . 63
6.4 Experiments 64
6.4.1 Function Minimisation 64
6.4.2 Catching a Ball 67
6.5 Discussion of Results 69

7 SEQUENTIAL FEATURE SELECTION 73
7.1 Introduction 73
7.2 Background 74
7.3 Sequential Online Feature Selection 76
7.3.1 General Idea 76
7.3.2 Application Scenarios 77
7.3.3 Additional Notation 78
7.3.4 Sequential Classification 78
7.3.5 Classifier State Representation 79
7.3.6 Classification as POMDP 81
7.3.7 Action Selection without Replacement 82
7.3.8 Solving the POMDP 83
7.4 Learning when to Stop 84
7.4.1 Potential Stopping Criteria 85
7.4.2 Integrated Classification 87
7.5 Experiments 89
7.5.1 Toy Example I - Shapes 90
7.5.2 Toy Example II - Cube 91
7.5.3 Handwritten MNIST digit classification 96
7.5.4 Diabetes Dataset with Naive Bayes Classification . 97
7.5.5 Integrated Classification on Cube Dataset 98
7.6 Discussion of Results 100

8 CONTEXT LEARNING 105
8.1 Introduction . . . 105
8.1.1 Spatial Context — Some Introductory Thoughts . . 105
8.1.2 Temporal Context . . . 108
8.1.3 How to Learn the Context . . . 110
8.2 Context Learning Framework . . . 112
8.2.1 Discrete State Context Learning . . . 112
8.2.2 Differentiation to Hidden Markov Models . . . 114
8.2.3 Finite State Machines and Regular Expressions . . 115
8.3 Experiments . . . 117
8.3.1 T-Maze . . . 117
8.3.2 Digits . . . 121
8.3.3 6-Bit Parity . . . 126
8.4 Discussion . . . 130
8.4.1 Context Representation . . . 130
8.4.2 Sub-Sequence Classification . . . 131

IV Conclusion 135

9 DISCUSSION 137

10 FUTURE WORK 141
10.1 State-Dependent Exploration . . . 141
10.2 Sequential Online Feature Selection . . . 142
10.3 Context Learning . . . 143
10.4 Combining SOFS with Context Learning . . . 145

BIBLIOGRAPHY 147

Part I

Overview

1 INTRODUCTION

"In order to understand recursion,
one must first understand recursion."

— Anonymous

Recognising patterns and predicting events based on previous observations is one of the main goals of Machine Learning. Because computer processing power doubles roughly every 18 months — the rule known as Moore's Law — these techniques have now found their way beyond research labs and into consumer electronics, social networks and the world wide web. They are used in credit card fraud detection, face recognition and product recommender systems, to name just a few examples. The digital age also brings with it an ever-growing amount of processable data, and trends over recent decades show that this rapid increase is, in fact, far outpacing Moore's Law. Although constant progress is being made in terms of developing new algorithms and refining methods to improve prediction results, the growth of data — and with it the redundancy of information — is starting to affect performance of such algorithms. This means that we cannot just rely on faster computers and continue to use brute force to crunch all available data regardless of its usefulness. We need to find other, smarter, ways to keep up with this "data explosion".

Our brains do a tremendous job of ignoring noise and focusing on the important and useful bits of information at each given moment. One of the core research questions that this thesis will attempt to answer is therefore:

> *How can Machine Learning methods overcome the data processing bottleneck, focusing on relevant data and ignoring noise and useless information? How can we apply the concepts of human attention and selective cognition to learning algorithms?*

This thesis addresses the question by training an attentive Reinforcement Learning controller that over time learns which portions of data seem relevant based on the current state of the underlying supervised learning method. It selects small chunks of data from a bigger sample that in the past were relevant to solving the supervised task. In doing

so, this approach turns a static classification problem with a large input dimensionality into a filtered sequence of relevant and less noisy lower-dimensional data.

While this may address the processing bottleneck problem, it raises a different issue, one that stems from the fact that sequence learning is intrinsically much harder to solve than classical static prediction. Sequential data contains temporal dependencies, for which the assumption of identical and independently distributed samples no longer holds. Many existing solutions to process sequence data have serious limitations in their abilities to store information over a long time, which leads to a second research question:

> *How can we learn an explicit representation of context independent of sequence length? Are there any alternatives to sequence learning that do not suffer from memory limitations?*

Again, Reinforcement Learning offers a solution to this problem. In this thesis, we lay the groundwork and demonstrate first results of learning an explicit contextual state during sequence processing, a context in which the current sample can be uniquely evaluated. We also show the close relationship between Context Learning and automata theory, deriving finite state machines as a special case of Context Learning.

Traditionally, Reinforcement Learning defines an environment in which agents optimise their behaviour over time based on sparse feedback. It is commonly used for control tasks in robotics, scheduling problems or game play. By applying it to general supervised learning tasks, however, we were able to create novel hybrid learning systems. Another, orthogonal goal of this thesis was to explore the possibilities of such hybrid algorithms that integrate Reinforcement Learning controllers into existing methods to solve supervised learning problems. The final question for this thesis is hence:

> *Can state-of-the-art Supervised Learning algorithms be improved by infusing them with Reinforcement Learning? And will the resulting hybrid algorithms perform qualitatively and/or quantitatively better on established benchmarks than the standard algorithms alone?*

These are the driving questions that this thesis endeavours to answer, and we will come back to them in the Discussion in Chapter 9 (page 137).

2 OUTLINE

This thesis is grouped into 4 major parts: Overview, Literature Review, Methods and Experiments, and Conclusion.

I OVERVIEW
The Overview contains Chapters 1–3. The thesis is introduced in Chapter 1, addressing the scientific problem and posing the key research questions. Chapter 2 (this chapter) gives a brief outline and structure of the thesis. Chapter 3 then details the scientific and technical contributions of this thesis and lists previously published work.

II LITERATURE REVIEW
The Literature Review consists of Chapters 4 and 5. Chapter 4 discusses existing literature and state-of-the-art of relevant function approximation (FA) methods. Due to its broad scope, this chapter only contains a coarse overview of general FA and then focuses on the need for FA in Reinforcement Learning (RL), followed by some popular FA techniques commonly used in RL. In Chapter 5, we then describe the current state in the field of Reinforcement Learning and its various sub-fields. After a formal definition and introduction of the notation, different types of RL algorithms both for discrete and continuous state and action spaces are discussed. The chapter ends with a review of RL exploration techniques.

III METHODS AND EXPERIMENTS
This part comprises the main contributions of this thesis, presented in Chapters 6–8. Chapter 6 introduces a novel exploration technique for Policy Gradient (PG) methods, called State-Dependent Exploration (SDE), that combines the benefits of both parameter-based exploration and direct action perturbation. We derive SDE theoretically and evaluate it on several benchmark tasks including a complex robotics simulation. Chapter 7 presents Sequential Online Feature Selection (SOFS), a novel algorithm that learns which features are most informative at any given time under a supervised classification directive. An RL-based attention mechanism guides the feature selection process online while concurrently minimising data consumption. Experiments on handwritten digits and medical data demonstrate its usefulness and superiority over existing methods. Chapter 8 then reveals Context Learning, a novel concept for sequential supervised learning that uses RL to build con-

text throughout a sequence in which current information is processed. After introducing the key idea, its relations to Automata Theory are discussed and differences to Hidden Markov Models are pointed out. In experiments, Context Learning shows great improvement over other deep memory algorithms both in convergence speed and memory depth.

IV CONCLUSION
The Conclusion includes Chapters 9 and 10. Where each chapter from Part III has their own discussion of results, Chapter 9 ties the individual topics together and concludes this thesis with pointers to human cognitive capabilities. Finally, directions for future research building upon this work are discussed in Chapter 10.

3 CONTRIBUTIONS

This thesis is the result of over six years of research, most of which was carried out at the Technische Universität München, Germany, with some parts being developed during a research period at the University of Western Australia in Perth, Australia. Below is a list of scientific (Section 3.1) and technical (Section 3.2) contributions that this dissertation adds to the current state of science.

3.1 SCIENTIFIC CONTRIBUTIONS

The main body of this thesis (for the most part Chapters 6–8) builds upon a number of already published, peer-reviewed publications in international journals and conference proceedings, which are listed below, and includes text passages and figures from these publications. I'd like to point out that most of these papers are based on joint work with my colleagues, as indicated by the list of authors. For sake of completeness, publications that I substantially contributed to are included in the list below, even if I am not the main (first) author on some.

Chapter 6 "State-Dependent Exploration", and the idea of parameter exploring RL algorithms is a joint work, mostly with Frank Sehnke, Martin Felder and Christian Osendorfer and based on the following previously published papers:

- T. Rückstieß, F. Sehnke, T. Schaul, D. Wierstra, S. Yi, J. Schmidhuber. **Exploring Parameter Space in Reinforcement Learning**. *Paladyn Journal of Behavioral Robotics, 1(1):14–24*, 2010.
- T. Rückstieß, M. Felder, and J. Schmidhuber. **State-Dependent Exploration for Policy Gradient Methods**. *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2008.
- F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. **Policy Gradients with Parameter-Based Exploration for Control**. *Proceedings of the International Conference on Artificial Neural Networks ICANN*, 2008.

- F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. **Parameter-Exploring Policy Gradients**. *Neural Networks, 23(4):551–559*, 2010.
- T. Rückstieß, M. Felder, F. Sehnke, and J. Schmidhuber. **Robot learning with State-Dependent Exploration**. *1st International Workshop on Cognition for Technical Systems*, 2008.
- F. Sehnke, T. Rückstieß, and J. Schmidhuber. **Parametric Policy Gradients for Robotics**. *1st International Workshop on Cognition for Technical Systems*, 2008.

Chapter 7 "Sequential Feature Selection", is a joint work and based on fruitful discussions with Christian Osendorfer, Justin Bayer and others. The results (except for Chapter 7.4, which presents new, unpublished findings) were published in the following papers:

- T. Rückstieß, C. Osendorfer, P. van der Smagt. **Minimizing Data Consumption with Sequential Online Feature Selection**. *Journal of Machine Learning and Cybernetics*, 2012.
- T. Rückstieß, C. Osendorfer, P. van der Smagt. **Sequential Feature Selection for Classification**. *Proceedings of the Australasian Conference on Artificial Intelligence*, 2011.

In addition to the pre-published material, Chapter 8 "Context Learning" contains unpublished ideas and results at the time of writing of this thesis that would warrant a scientific contribution. It is planned to release the findings of Chapter 8 as a separate publication at a later stage.

3.2 TECHNICAL CONTRIBUTIONS

In addition to the publications mentioned above, several software tools and libraries were developed and used to conduct the experiments and verify the hypotheses and algorithms of Part III. Of these software tools, two were successfully used beyond this thesis and thus published in journals for the benefit of other researchers and students.

PyBrain

PyBrain (Schaul et al., 2010; Kovacs and Egginton, 2011) is an open source software library for machine learning, written in Python. Its

purpose is to provide an easy-to-use flexible platform for Neural Networks, Reinforcement Learning algorithms, optimisation methods and other well-known supervised and unsupervised machine learning algorithms. PyBrain was jointly created by two groups of researchers at the Technische Universität München, Germany and IDSIA Institute for Artificial Intelligence in Lugano, Switzerland. PyBrain has an active community and a mailing list with over 1000 users to date. My contributions to PyBrain include many of the Reinforcement Learning algorithms, the core dataset structure and various additional internal parts, as well as the development and maintenance of the PyBrain website `http://www.pybrain.org`, the PyBrain logo and active and ongoing support for the mailing list.

The main publication describing its features and giving an overview of its structure is:

- T. Schaul, J. Bayer, D. Wierstra, S. Yi, M. Felder, F. Sehnke, T. Rückstieß, J. Schmidhuber. **PyBrain**. *Journal of Machine Learning Research, 11:743–746*, 2010.

Python Experiment Suite

The Python Experiment Suite (PyExpSuite) is an open source software tool written in Python, that supports scientists, engineers and others to conduct automated generic software experiments on a larger scale with numerous features: parameter ranges and combinations can be evaluated automatically, where different experiment architectures (e.g. grid search) are available. The suite also takes care of logging results into files and can handle experiment interruption and continuation, for instance in circumstances where the process is terminated due to power failure. It also supports execution on multiple cores and contains a convenient Python interface to retrieve the stored results. Configuration files ease the setup of complex experiments without modifying code and various run-time options allow for a variety of use cases.

PyExpSuite was implemented to facilitate the repeated execution of the experiments in this thesis and to guarantee the reproducibility and correctness of their results. The publications describing PyExpSuite and its source code are:

- T. Rückstieß, J. Schmidhuber. **A Python Experiment Suite**. *The Python Papers Journal 6(1):2*, 2011.
- T. Rückstieß, J. Schmidhuber. **Python Experiment Suite Implementation**. *The Python Papers Source Codes, 2:4*, 2011.

Part II

Literature Review

4 FUNCTION APPROXIMATION

4.1 GENERAL

Function approximation refers to the selection of a function f from a well-defined class of possible functions which most closely matches a target function g. In the machine learning domain, the target function g is usually not known but a finite number of values $x_i \in X$ from the domain of g and their mappings $g(x_i) \in Y$ are known (called a *dataset*). It is then the goal to find a function f, which interpolates g in the range of x but also extrapolates on values lying outside the boundaries of x.

As there is an infinite number of such functions available that exactly match $g(x_i)$ for all x_i, another desirable property for function approximation is that of a simple model (or, in other words, a model of "low complexity"). For example: if a dataset of n points is available, we can always derive a polynomial of degree $n-1$ to perfectly match those n points. But this is often not desirable as we would *over-fit* the data and interpolation for new points would result in large prediction error. A lower degree polynomial would most likely not match the dataset perfectly but it would yield much better results for new, unknown points because it has a lower complexity. Conversely, increasing the number of samples in a dataset (while keeping the model complexity constant) would also reduce the chance of over-fitting. Therefore, one could say that one of the universal truths of machine learning is: the more data available, the better the resulting model.

Function approximation is also a useful tool to overcome noisy data. Most real-world problems suffer from noisy inputs, due to imperfect sensors or non-deterministic processes. Function approximation can not only abstract over such noisy data by averaging out the noise, but noise can also be proactively modelled and estimated under some stochastic FA methods.

Function approximation in machine learning is the key for making predictions about unseen (or unknown) data. In almost every aspect of machine learning, function approximation plays an integral part in representing knowledge based on past experiences or observations.

The field of general function approximation is too large to cover here in all detail. In the following sections we will focus on its relation-

ship and application to Reinforcement Learning and introduce some of the most popular function approximation techniques used for Reinforcement Learning, in particular those that will be used in later chapters of this thesis.

4.2 FUNCTION APPROXIMATION IN REINFORCEMENT LEARNING

Function approximation in the reinforcement learning context can be useful in a number of different ways. Tabular reinforcement learning covers problems with finite[1], discrete states and actions. If the problem requires continuous states or continuous actions (or even both), as it is often the case in robot control tasks, function approximation is required to cover the infinite number of cases. The function approximator would then replace the value table and return an approximate value for a state/action pair as input. While many of the strong convergence proofs of tabular RL algorithms do not hold for the continuous case (Sutton, 1999), in practise this approach often works quite well.

Function approximation is not just needed for continuous state/action spaces, it is also very useful in the discrete state/action problems as it allows the transfer of knowledge to similar states and actions. This technique can be applied as long as the states/actions have a distance metric defined on them, and similar states/actions (in terms of behaviour) lie close by in their space. Then a function approximator can be used to generalise over the executed states/actions and assign reward credit to similar neighbouring states/actions.

Function approximation also plays and important role in direct, model-free RL. Here, instead of learning values for state-action pairs, a differentiable function approximator maps directly from states to actions and errors are back-propagated through the function approximator to change its parameters to values that yield more rewarding actions in the future.

The following chapters discuss some of the function approximation algorithms most often found in Reinforcement Learning and in particular those required for later chapters in this thesis.

1 here, "finite" does not mean theoretically countably many, but finite in a practical sense. Tabular RL algorithms with more than a few hundred states or actions will require a very long time to converge.

4.3 LINEAR REGRESSION

Linear Regression is one of the most basic optimisation algorithms. As the name suggests, data is modeled under a linear relationship between input and model parameters, mapping a k-dimensional input to a 1-dimensional target. Because of the linear nature, the mapping can be calculated as a simple matrix multiplication and is therefore very fast and efficient. The parameters to be trained correspond to the coefficient values θ_i, which can be calculated with the *Moore-Penrose* pseudo-inverse.

Assuming a dataset of n inputs $X = \{(x_{i1}, x_{i2}, \dots, x_{ik})\}_{i=1}^{n}$ with $x_i \in \mathbb{R}^k$, targets $y = \{y_i\}_{i=1}^{n}$ and noise terms ϵ_i, we are interested in the regression coefficients $(\theta_1, \dots, \theta_k)$ to solve the linear equation system given as

$$y = X\theta + \epsilon = \begin{bmatrix} x_{11} & \dots & x_{1k} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nk} \end{bmatrix} \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_k \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}. \tag{4.1}$$

As the equation system is usually overdetermined, i.e., we have $n > k$, a least-squares approach is used to minimise the sum of squares of all errors ϵ_i. The regression coefficients θ_i can be estimated by calculating

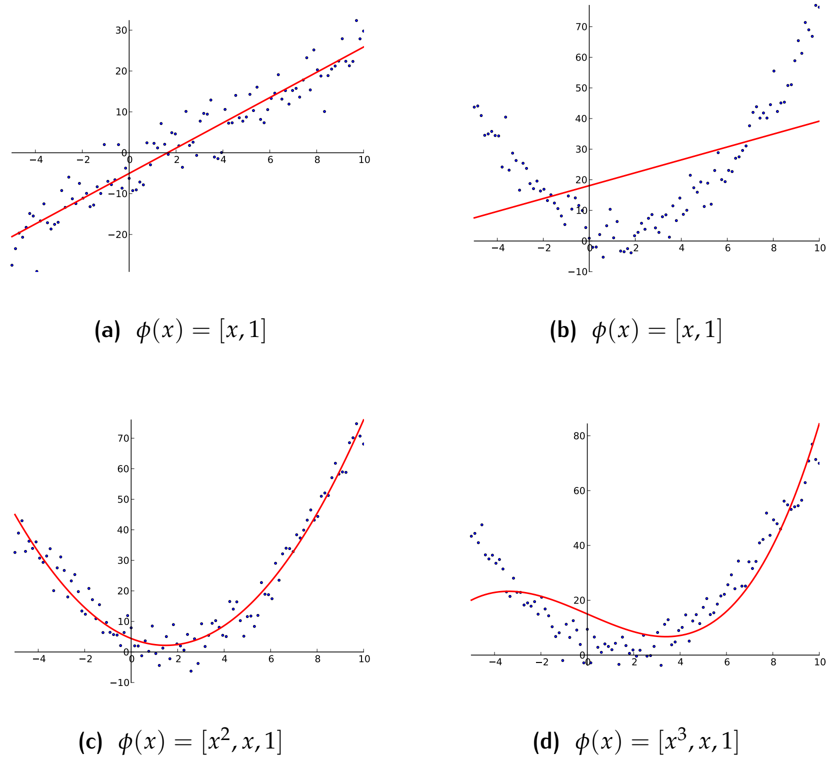
$$\theta = (X^T X)^{-1} X^T y = X^+ y, \tag{4.2}$$

where X^+ is called the *Moore-Penrose pseudo-inverse* (or simply *pseudo-inverse*) of X. It is common practise to extend X with a constant column vector of 1 (as a $(k+1)$th dimension). This adds an additional element θ_{k+1} to the coefficient vector θ, called *intercept*, and acts as an offset to y, allowing the regression function to return a non-zero value for $y = f_\theta(x)$, even if $x_i = 0 \ \forall i$. Prediction of an unseen input x is then calculated as

$$y = f_\theta(x) = x^T\theta = \sum_{i=1}^{k+1} x_i\theta_i, \tag{4.3}$$

with x being concatenated with a constant 1 prior to calculation.

While both the calculation of the parameters θ from a known dataset (X, y) and prediction of unseen inputs can be calculated very efficiently, one big drawback of linear regression is its limitation to linear dependencies between input and target variables. Figure 1(a) and (b) illustrate examples on linear and quadratic data, respectively. While the result in Figure 1(a) looks alright, the regression in 1(b) is clearly not what was intended. One thing to note is that the linear assumption only needs to hold for θ, while the inputs x can be non-linear. In fact, linear regression



(a) $\phi(x) = [x, 1]$ **(b)** $\phi(x) = [x, 1]$

(c) $\phi(x) = [x^2, x, 1]$ **(d)** $\phi(x) = [x^3, x, 1]$

Figure 1: Linear regression examples with different basis functions. Identity basis function plus added constant term works well for linear data (a) but fails on more complex data (b). Using the correct basis functions, ideally from the same family of functions that the original data stems from, leads to good fits (c). With poorly choosen basis functions, the fit is usually suboptimal (d).

is often formulated in terms of (non-linear) basis functions $\phi(x)$, where $\phi(x)$ takes a vector x and maps it into a (usually) higher-dimensional space. So instead of the equation system given in Eqn. (4.1), we can solve the following:

$$y = \Phi\theta + \epsilon = \begin{bmatrix} \phi(x_1)^T \\ \vdots \\ \phi(x_n)^T \end{bmatrix} \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_p \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}, \tag{4.4}$$

with $\phi(x) \in \mathbb{R}^p$ and $\Phi = [\phi(x_i)]_{i=1}^n$ is an $n \times p$ matrix, called the *design matrix*. Choosing a good design matrix requires knowledge of the data, ideally the basis function maps x into the same space as the data originally came from. Figure 1(c) and (d) show the fit for a good and suboptimal choice of basis functions.

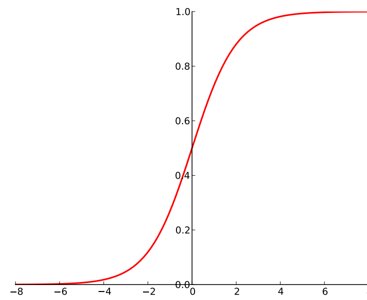


Figure 2: The logistic function is a sigmoid function, mapping continuous values from the domain of real numbers to the range [0, 1].

Linear regression is of use when data is very high-dimensional and efficient mapping is required or the family of functions that the data is drawn from is known. For low-dimensional but complex, non-linear data, other function approximation methods are prefered.

4.4 LOGISTIC AND MULTINOMIAL REGRESSION

Despite their names, both Logistic and Multinomial Regression are actually classification methods, where the outcome is no longer a continuous variable but rather a categorical class (i.e., one of a finite number of states). Logistic Regression is a binary classification algorithm, separating the input patterns in one of two output classes. Multinomial Regression is the extension of this principle to multiple classes. In either version, the probability of an input pattern belonging to one of the resulting classes is modelled.

Below we will summarise the estimation of the parameters for Logistic Regression (see e.g. MacKay (2003) or Bishop (2006) for a detailed description). The logistic function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.5}$$

which results in a sigmoid shape, see Fig. 2. The fact that this function's derivative with respect to x can be expressed in terms of the logistic function itself makes it computationally more efficient:

$$\frac{\partial \sigma}{\partial x} = \sigma(1 - \sigma). \tag{4.6}$$

Maximum Likelihood estimation is used to solve for the parameters of the model, leading to an error gradient

$$\nabla E(\theta) = \sum_{i=1}^{n} (x_i - y_i)\,\phi_i \tag{4.7}$$

where we use the same notation as before, denoting the inputs with x and the targets with y, represented as 0 and 1, and basis functions ϕ. Eqn. (4.7) can be applied in iterative manner, updating the parameters θ for each presentation of input/target tuple. The output of the model is the estimated probability of the input pattern belonging to the class. It is common to use a threshold at 0.5 to distinguish between the two classes. For Multinomial Regression, a Softmax transformation is used to determine the winning class, where the output vector is

$$y_i(\phi) = \frac{\exp(a_i)}{\sum_k \exp(a_k)}, \quad \text{with } a_i = \theta_i^T \phi. \tag{4.8}$$

4.5 NEURAL NETWORKS

Neural Networks (or *Artificial Neural Networks*, to distinguish them from their biological counterparts) consist of several layers (usually input, hidden and output layer for normal feed-forward networks) of processing units (called *neurons*), which process information from the layer below and propagate it to the layer above. Each connection between two neurons is called a *weight*. Many architectures are possible, and neural networks are usually separated in two classes: those with *recurrent* connections, i.e. connections that connect neurons from the same layer with each other or with neurons in lower layers, and those without such connections. Members of the latter class are called *feed-forward networks* because the information only flows in one direction through the network, from the bottom to the top.

Recurrent neural networks on the other hand contain circular information flow. This property allows the network to have a memory because circular connections can store information over several steps. One such class, that is known to store information exceptionally well, is *Long Short-term Memory* (LSTM) networks (Hochreiter and Schmidhuber, 1997). In the sections below, we will introduce feed-forward networks and illustrate the training algorithm as well as briefly review LSTMs as an example for recurrent networks.

A thorough overview of the the history of Neural Networks and Backpropagation is given by Schmidhuber (2015). Some of the milestones are described in the sections below.

4.5.1 Feed-Forward Neural Networks

The first non-static, learning Neural Networks were published as early as 1958 (Rosenblatt, 1958) and gained popularity in the following decades (Widrow and Hoff, 1962; Narendra and Thathatchar, 1974). Chained application of the gradient descent method (Hadamard, 1908) to minimise errors in nonlinear NN-like systems was discussed as early as 1960 (Kelley, 1960; Bryson, 1961), with an elegant derivation published solely based on the chain rule shortly after (Dreyfus, 1962).

Linnainmaa (1970) first described the algorithm now known as *back-propagation* in its modern form for discrete sparse networks and provided a Fortran reference implementation. Dreyfus (1973) applied back-propagation to adopt control paramters. Its explicit application to neural networks was first published by Werbos (1981) with references to related thoughts in his Ph.D. thesis (Werbos, 1974). Rumelhart et al. (1986) describes how back-propagation can be used to learn internal representations.

Feed-forward neural networks with at least one hidden layer have long been shown to be universal function approximators (Kolmogorov, 1965; Funahashi, 1989; Cybenko, 1989; Maxwell and White, 1989; Bishop, 1995), being able to approximate any function, even non-linear, given enough training data. Input patterns x are presented to the lowest layer. Each neuron in higher layers has access to all the neurons in the layer below, multiplying each input with the corresponding weight, summing up all the weighted inputs, applying an *activation function* f_{act} to the sum and passing the result to the next layer (see Figure 3). The highest layer outputs the network's prediction for the given input.

To train a feed-forward neural network, the error between an actual output y and the desired target d is calculated and *back-propagated* through the output layer down to the input layer. By repeatedly applying the chain rule, one can calculate how big each weight's contribution to the error was and by what amount it needs to be changed to compensate for the error. A learning rate $0 < \alpha \leq 1$ ensures slow but steady convergence during the gradient descent.

Back-Propagation Algorithm in Detail

For neuron j, a_j denotes the net input to the neuron, z_j denotes the neuron's output, e_j is the error of neuron j compared to the teaching input

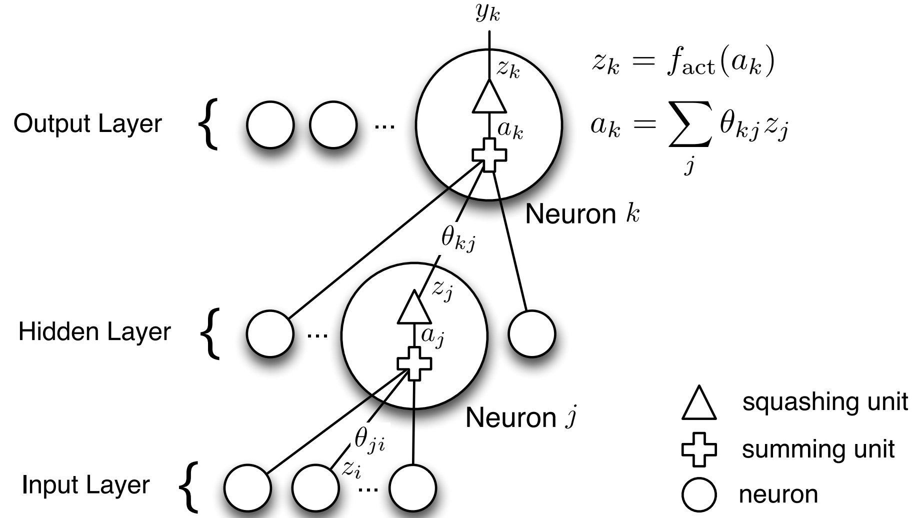


Figure 3: Illustration of a Feed-Forward Neural Network. From bottom to top: each neuron from a lower layer connects to every neuron in the next layer. Inputs z_i are multiplied with weights θ_{ji} and summed up for each neuron in the upper layer, resulting in the net input a_j. The net input is then passed through an activation function f_{act} which yields the neurons activation z_j that is passed to the next layer.

d_j. E is the total error of the network and $\triangle w_{ji}$ denotes the calculated change for weight w_{ji} from neuron i to j.

$$a_j = \sum_i w_{ji} z_i \tag{4.9}$$

$$z_j = f_{\text{act}}(a_j) \tag{4.10}$$

$$e_j = y_j - d_j \tag{4.11}$$

$$E = \frac{1}{2} \sum_j (e_j)^2 \tag{4.12}$$

$$\triangle w_{ji} = \alpha \frac{\partial E}{\partial w_{ji}} = \alpha \underbrace{\frac{\partial E}{\partial a_j}}_{:=\delta_j} \underbrace{\frac{\partial a_j}{\partial w_{ji}}}_{=z_i} = \alpha\, \delta_j\, z_i \tag{4.13}$$

To calculate the delta terms for output units, we start with the term denoted δ_j in (4.13) and apply the chain rule to receive

$$\delta_j = \frac{\partial E}{\partial a_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial a_j}. \tag{4.14}$$

Substituting (4.11) in (4.12) and differentiating with respect to y_j, we obtain

$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j}\left(\frac{1}{2}\sum_k (y_k - d_k)^2\right) = (y_j - d_j) \tag{4.15}$$

with teaching input d_j for an output neuron j. Now we can substitute z by y in (4.10) (the outputs of output neurons are identical to the net output) and rewrite the derivative as

$$\frac{\partial y_j}{\partial a_j} = f'(a_j). \tag{4.16}$$

Finally, substituting (4.15) and (4.16) into (4.14) yields

$$\delta_j = f'(a_j) \cdot (y_j - d_j). \tag{4.17}$$

To calculate the δ terms for input and hidden units, we start with

$$\delta_j = \frac{\partial E}{\partial a_j} = \sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{4.18}$$

and apply the chain rule for partial derivatives. The first factor in the sum is defined as δ_k, whereas the second one can be obtained by substituting equation (4.10) into (4.9) and differentiating:

$$a_k = \sum_i w_{ki} f(a_i) \tag{4.19}$$

$$\frac{\partial a_k}{\partial a_j} = w_{kj} f'(a_j). \tag{4.20}$$

Substituting (4.19) and (4.20) in (4.18), we can write

$$\delta_j = \sum_k \delta_k w_{kj} f'(a_j) = f'(a_j) \sum_k \delta_k w_{kj}. \tag{4.21}$$

To apply the weight changes to each neuron in the output layer, we can calculate the deltas $\triangle w_{ji}$ according to

$$\triangle w_{ji} = \alpha\, \delta_j z_i, \quad \delta_j = f'(a_j) \cdot (y_j - d_j)$$

with learning rate α.

For neurons in the input and hidden layer, we apply the following rule recursively back from the output to the input layer, knowing all δ_k from previous neurons already:

$$\triangle w_{ji} = \alpha\, \delta_j z_i, \quad \delta_j = f'(a_j) \sum_k \delta_k w_{kj}.$$

4.5.2 Recurrent Neural Networks and LSTMs

Figure 4: Compound LSTM cell. A LSTM recurrent neural network is comprised of these cells, each containing an input unit (blue), several gate units (green) that control how information can enter and leave the cell and a linear unit at the core called a Constant Error Carousel (CEC). The CEC is connected to itself and any signal "trapped" within the CEC can cycle the carousel indefinitely without getting squashed by an activation function.

Recurrent neural networks (RNN) differ from their non-recurrent (feed-forward) counterparts in that they allow lateral connections from one layer to itself or even back to previous layers, opposed to feed-forward network layers, which strictly connect to higher layers. This small but important difference allows recurrent networks to store information over time and makes them capable of processing temporal signals. The idea is that a temporal sequence is passed into the network time step by time step, and information from past activations can cycle in loops in the hidden layer until it is needed at a later time.

Recurrent networks come in different flavours each with certain restrictions on their architecture. Jordan networks (Jordan, 1986), for example, only allow connections from the output layer back to the input layer as so-called "context neurons", while Elman networks (Elman, 1990) are a variation of Jordan networks that connect only from the hidden layer back to the input layer. Another, more recent type of recurrent networks are Long Short Term Memory (LSTM) networks (Hochreiter and

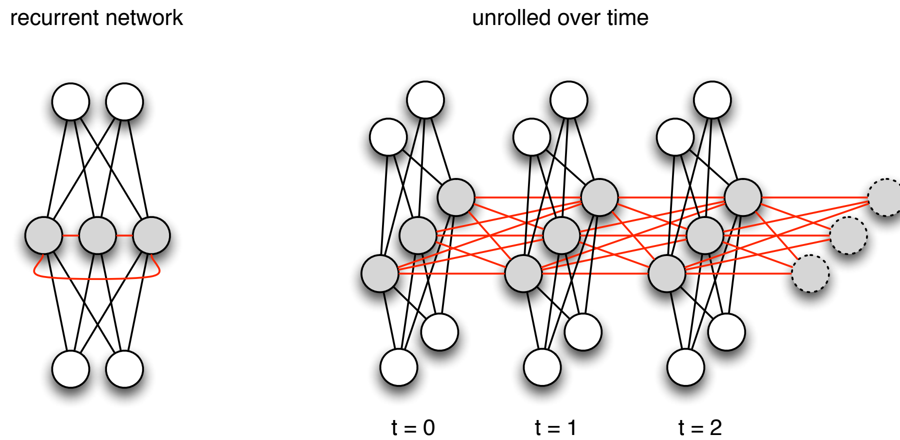


Figure 5: Recurrent neural network (left) with shaded hidden nodes and recurrent connections in red and the same network unrolled in time (right). The hidden nodes fully connect to a a temporal copy of themselves for each time step.

Schmidhuber, 1997) that use compound cells as neurons with a certain fixed wiring inside these cells, illustrated in Figure 4. They have the ability of storing information indefinitely while "plain" RNNs often forget any information after only a small number of time steps, due to the vanishing gradient problem. LSTMs are also known for their ability to count and learn precise timings of temporal events (Gers et al., 2002; Hüsken and Stagge, 2003).

Applying back-propagation to recurrent networks is straight forward, and the most prominent purely supervised algorithms are known as Back-Propagation Through Time (BPTT) (Werbos, 1988, 1990), Real-Time Recurrent Learning (RTRL) (Williams and Zipser, 1989) and a combination of both of these methods (Schmidhuber, 1992). The basic idea of all these training algorithms is the same as the original Back-Propagation algorithm: the gradient of the error with respect to each weight is calculated and the weights are changed in direction of minimised error. For recurrent networks, one can imagine an "unrolled" network in time (Figure 5) that for each timestap has a copy of the complete architecture of the equivalent recurrent network without cycles. The recurrent weights connect each of these copies with the next. By eliminating the cycles this way, we create a feed-forward network that can be trained with a slightly modified Back-Propagation algorithm. The recurrent weights, however, are only virtual shared weights and changes to one of them affects all copies.

However, deep network architectures generally suffer from a problem called the vanishing gradient. The repeated application of typical activation functions used for neural networks causes the error to shrink below

(or explode beyond) computer floating point accuracy after only a few layers of neurons (Hochreiter, 1991; Bengio et al., 1994). One popular way to avoid the issue, which is still relevant today in its basic form, is to train a stack of single-layer networks in unsupervised fashion as Autoencoders (Ballard, 1987; Hinton and Salakhutdinov, 2006; Hinton et al., 2006a,b), learning to compress information and re-creating it from the compressed representation. Each additional layer learns to encode some of the information that the previous layer was unable to learn. This stack of pre-trained networks is then further fine-tuned with back-propagation.

4.6 LOCALLY WEIGHTED PROJECTION REGRESSION

Locally Weighted Projection Regression (LWPR) is a non-parametric non-linear function approximation method, that uses locally linear models at its core which are spanned in selected directions in the input space (Vijayakumar and Schaal, 2000). The algorithm is particularly well-suited for large amounts of data and high-dimensional input spaces with the data ideally lying on a lower-dimensional manifold. LWPR performs a weighted variant of the Partial Least Squares (PLS) (Wold, 1975) regression algorithm in order to reduce the dimensionality of the data. LWPR is very fast to compute as it has a linear computational complexity in the number of input samples and works in an incremental online fashion that adapts over time, serving as a fast method for incremental value function approximation.

4.7 SEQUENCE LEARNING

This section provides some remarks about Sequence Learning in general, and how to deal with sequential data. This topic is closely related to recurrent neural network algorithms, see Section 4.5.2.

One of the main difficulties of sequence learning are temporal long-term (non-Markovian) dependencies (Sun and Giles, 2001). Many Machine Learning algorithms cannot cope with these long-term dependencies, where a state reaching back a long time has an effect on a state appearing much later in the sequence. It is important that such dependencies can be discovered, for example by providing the algorithm with long-term accessible memory (see LSTM networks in section 4.5.2). Another problem is hierarchy in sequences, where a sequence consists of sub-sequences, which themselves can consist of smaller sub-sequences, and

so forth. Discovering such hierarchies would be helpful for the learning task, but is intrinsically difficult to achieve.

There are several types of sequence learning problems, due to the time-dependent nature of data. Most commonly, a distinction exists between sequence classification (mapping a full sequence to one of n classes), sequence prediction (predicting the next element in a sequence based on all previous elements) and sequence generation (creating a sequence of outputs based on a sequence of inputs). All problems can be generalised into a single problem definition, if we allow the class labels to be numerical values and introduce an *importance* factor for each time step. Figure 6 illustrates this schematically.

The importance factor indicates how important it is to get the answer right for the current time step. This factor is multiplied with the error during back-propagation, therefore an importance of 0.0 will cause the error not to have any influence on the weights for that time step. For sequence classification, the importance factor is 0.0 for the full sequence length except for the last element, where it is 1.0. Sequence prediction and sequence generation are handled by having an importance factor of 1.0 for the full sequence.

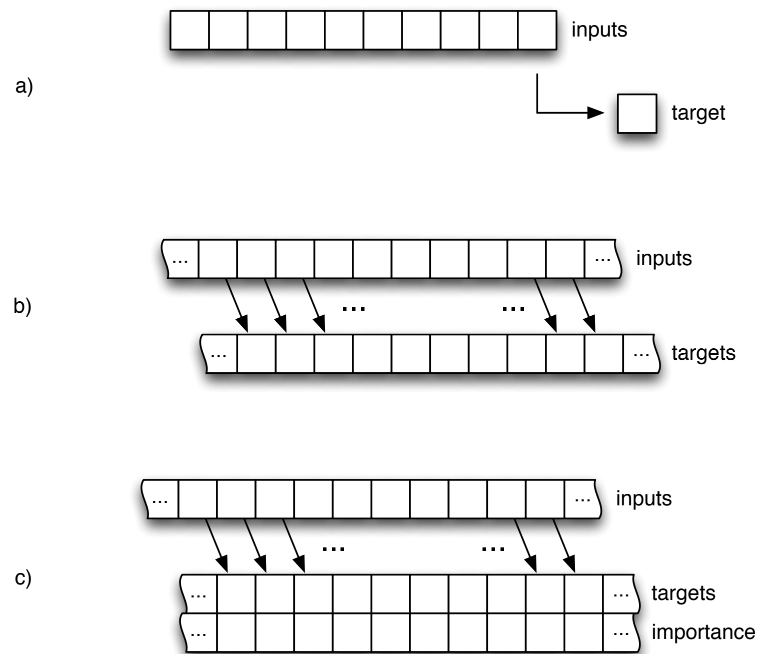


Figure 6: Different sequence learning types can all be handled with one unified framework. a) Sequence classification assigns a full sequence to a single class. b) Sequence prediction and sequence generation both require an output at each time step. c) With an importance factor for each time step, all three sequence learning problems can be mapped to the same framework. For classification, all but the last importance element are set to 0 and arbitrary targets (usually 0.0) can be used during the sequence. For prediction and generation, the importance is set to 1.0 throughout.

5 REINFORCEMENT LEARNING

5.1 GENERAL REMARKS AND NOTATION

Generally speaking, Reinforcement Learning (RL) aims to optimise an agent's behaviour in an environment over time.

The Reinforcement Learning framework is defined in a very general matter. The term *agent* is used for any kind of program, algorithm, robot or other system that can perceive inputs and react to them with outputs. We will call these inputs *states* and the outputs *actions*. The agent's behaviour—its mapping from states to actions—is called a *policy* and can be manipulated via feedback to the agent. This feedback is commonly called *reward* and can be perceived by the agent as a separate signal.

States can be actual states of a physical machine (e.g. on/off), an internal representation of a virtual environment, or abstract concepts. An agent could for example be in a state "I don't know what this object is". The same is true for actions: They can be very concrete low-level motor commands or high-level concepts, such as "going to lunch". Time steps do not have to refer to fixed intervals of real time, they can also be successive stages of decision making not related to a fixed time frame.

An *environment* is very similar to the concept of an agent. It, too, reacts to inputs with certain outputs. As the environment takes the antagonistic role to the agent's, it will use the agent's outputs (actions) as input and will feed its outputs back to the agent as states. The difference is that the environment's mapping from inputs to outputs cannot be influenced externally. Therefore, there is also no reward mechanism involved in the environment. Figure 7 illustrates the cycle of interaction between agent and environment.

Rewards can be seen as originating from an external source. In this case, the environment has a passive, reactive role and is considered not to have an intention or "plan" for the agent. However, it is not uncommon to have the rewards originate directly from the environment. This is certainly the case in real-life scenarios. Both versions are equivalent from the agent's perspective, which only cares about the amount of reward and not its source. Optimisation in the above statement refers to maximisation of received rewards over time.

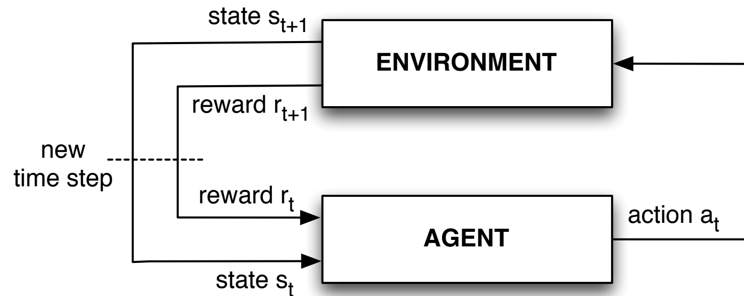


Figure 7: The Reinforcement Learning cycle. The agent receives the current state s_t from the environment and reacts with action a_t. This changes the environment to state s_{t+1} and causes a reward of r_{t+1}.

The concept of challenge/response in an agent can be found in supervised learning tasks as well. But unlike supervised learning, where the correct answer is provided during training for each challenge, in Reinforcement Learning the agent is not told the correct actions directly, but is only informed about how well it did, in terms of a scalar *reward* value, which can arrive with an unknown delay. Thus, the available information about the agent's performance is much more scarce and it is not immediately obvious how to utilise the scalar feedback to modify the policy. This question is what RL tries to answer.

Applications for Reinforcement Learning are manifold, reaching from robot control like flying (Abbeel et al., 2007), walking (Nakamura et al., 2007) or grasping (Montesano and Lopes, 2009); playing games like BackGammon (Tesauro, 1994), Go (Grüttner et al., 2010), or video games like Super Mario (Mohan and Laird, 2009); face recognition (Harandi et al., 2004); job scheduling (Zhang and Dietterich, 1995); financial trading (Moody and Saffell, 2001), to only name a few of the many areas. This list is by no means complete, but gives an idea of the broad variety of applications that Reinforcement Learning is being used for to solve real-world problems. Further applications are pointed out in the following sections where appropriate.

5.1.1 Formal Definition of Reinforcement Learning

Learning proceeds in a cycle of interactions between the agent and the environment. In time t, the agent observes a state $s_t \in \mathcal{S}$ from the environment, performs an action $a_t \in \mathcal{A}(s_t)$ and receives a reward $r_{t+1} \in \mathbb{R}$. Rewards can be given in every time step or sparsely (e.g. only when a goal state is reached), in which case $r_t = 0$ for $t \neq t_{\text{goal}}$. Rewards can further be probabilistic: the expected reward when transitioning from state s to s' via action a is denoted by $\mathcal{R}^a_{ss'}$. The environment

then determines the next state s_{t+1} where the probability of transitioning from state s to s' with action a is written as $\mathcal{P}^a_{ss'}$.

The term *history* denotes the concatenation of all encountered states, actions and rewards up to time t as $h_t = \{s_0, a_0, r_1, \ldots, s_{t-1}, a_{t-1}, r_t, s_t\}$. RL can be episodic, in which case at least one of the states in $\mathcal{S}$ is terminal and the histories have finite length. On the other hand, continuous RL (also called *infinite-horizon* RL) will have a history with infinite length. In either case, the objective of RL is to maximise the long term *return* R_t, which is defined as the (discounted) sum of future rewards, starting from time t:

$$R_t = a_\Sigma \sum_{k=0}^{T} a_D \, r_{t+k+1} \tag{5.1}$$

Here, $a_\Sigma = (1-\gamma)$, $a_D = \gamma^k$ for discounted (possibly continuous) tasks and $a_\Sigma = 1/T$, $a_D = 1$ for undiscounted (and thus necessarily episodic) tasks. For infinite-horizon tasks with $T = \infty$ that do not have a foreseeable end, $\gamma < 1$ prevents unbounded sums.

Actions are selected by a policy π that maps a history h_t to a probability of choosing an action: $\pi(h_t) = p(a_t|h_t)$. For deterministic policies, where $p(a'|h) = 1$ and $p(a|h) = 0 \; \forall a \in \mathcal{A} \setminus a'$, the function f_π represents the mapping from history to action: $f_\pi(h) = a'$.

Parameterised policies have a parameter vector θ through which the policy $\pi(h_t; \theta)$ (or $\pi_\theta(h_t)$ in short) can be manipulated. In this thesis all policies are parameterised, therefore the θ is sometimes omitted for clarity.

Often, the environment fulfils the Markov assumption, i.e. the probability of the next state depends only on the last observed state and the current action: $p(s_{t+1}|h_t) = p(s_{t+1}|s_t, a_t)$. Markovian environments summarise all relevant information from the past in their current state. A chess game is Markovian, because each constellation of pieces on a board contains all the information required to make the next move. It is not important how the pieces got there. A poker game, on the other hand, violates the Markov assumption. The agent needs to remember, how its opponent acted throughout the whole game, not just in the last round, to make the best decision. If the environment is Markovian, it has a stationary $\mathcal{P}^a_{ss'}$ and $\mathcal{R}^a_{ss'}$ and it is sufficient to consider Markovian policies that satisfy $p(a_t|h_t) = p(a_t|s_t)$. Unless stated otherwise, all described environments are assumed to behave Markovian.

Further, the expectation operator is denoted as $E\{\cdot\}$ and the sample mean is written as $\langle \cdot \rangle$.

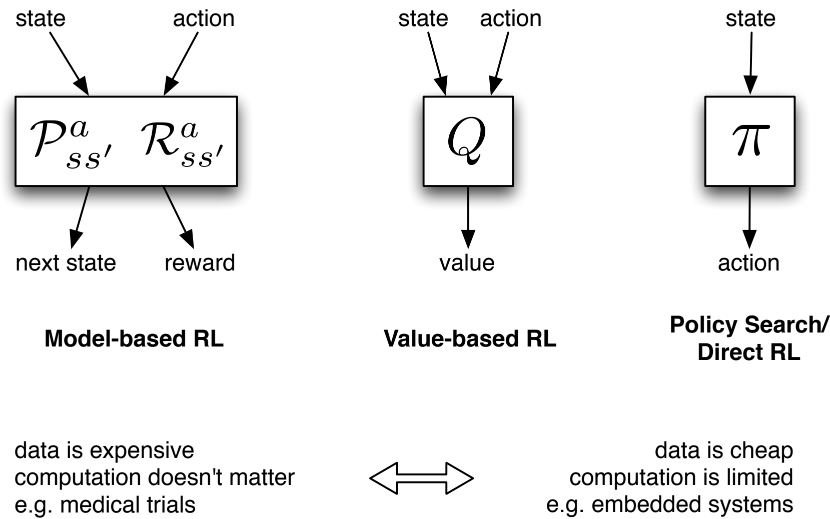


Figure 8: Different Reinforcement Learning categories.

In this thesis, the large set of Reinforcement Learning algorithms is clustered into three different sub-categories: *model-based*, *value-based*, and *policy search* algorithms (also called *direct RL* algorithms), mainly due to their different approach of handling data (illustrated in Fig. 8). This view on the field of Reinforcement Learning may not necessarily cover all existing algorithms, and neither should be seen as a strict taxonomy. Many hybrid methods and extensions exist, that might not fit well in either of these categories. For the work presented in this thesis, however, this distinction was chosen to better understand the presented research.

Direct RL maps states directly to actions via a parameterised policy function. During learning, the parameters are changed towards a policy that yields a better overall return. Value-based methods use a dual form, estimating a value for each state (or state-action pair), which describes how good it is to be in that situation. These algorithms make better use of the data than policy search methods, since they build an internal representation of the state-reward space. Model-based RL goes another step towards efficient data usage by employing state-transition probabilities and mean reward estimates to solve the problem efficiently, e.g. with dynamic programming approaches. If these state-transition probabilities are not known, they are estimated from experience.

These three categories of Reinforcement Learning algorithms are presented in more detail in Sections 5.2, 5.3 and 5.4.

5.2 MODEL-BASED REINFORCEMENT LEARNING

Model-based RL aims to estimate the transition probabilities $\mathcal{P}^a_{ss'} = p(s'|s,a)$ and the rewards $\mathcal{R}^a_{ss'} = E\{r|s,s',a\}$ going from state s to s' with action a. Having a model of the environment allows one to use direct or value-based techniques within the simulation of the environment, or even for dynamic programming solutions.

For fully known models, e.g. in form of a completely described Markov-Decision Process (MDP), one can use value iteration (Bellman, 1957) or policy iteration (Howard, 1960). For large or continuous state spaces, or where the underlying MDP model is not known, a solution is approximated by averaging over the collected samples, as the closed form computation is usually infeasible. In this thesis, we will therefore focus on value-based and direct RL methods instead.

5.3 VALUE-BASED REINFORCEMENT LEARNING

One approach to solve Reinforcement Learning problems as stated above is given by value-based methods. Value-based RL assumes Markovian environments and uses a *value function* to represent how "good" it is to be in a certain situation. There are a number of different types of value functions (Sutton and Barto, 1998), but here we will mostly deal with *action-value functions*, which map each combination of state and action to a Q-value. More precisely, an action-value function $Q^\pi(s,a)$ with respect to a policy π is a function of state s and action a, which describes the expected future return R_t if the agent starts in state s, takes action a and follows policy π thereafter:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{R_t|s_t = s, a_t = a\} \\ &= E_\pi\{a_\Sigma \sum_{k=0}^{T} a_D\, r_{t+k+1}|s_t = s, a_t = a\} \end{aligned} \tag{5.2}$$

Another type of value functions are *state-value functions*, which are only concerned about the current state, independent of the next action taken. They are defined as $V^\pi(s) = E_\pi\{R_t|s_t = s\}$. It is straight-forward to derive the state-value for any state given the action-values with $V^\pi(s) = Q^\pi(s,a),\ a \sim \pi(s)$. Figure 9 illustrates the difference between the two value functions.

With the introduction of value functions, policies can be compared to each other. A policy π is considered "better" than π' if the expected

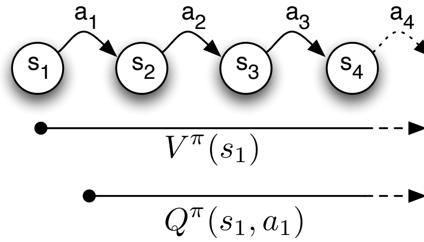


Figure 9: Difference between state-value functions and action-value functions. State-value function V^{π} considers a state as its starting point after which it follows policy π. Action-value function Q^{π} takes both the current state and the current action into consideration before it follows policy π, thus carrying more detailed information. $V^{\pi}(s)$ can always be derived from $Q^{\pi}(s, a)$, but not vice versa.

returns of π are higher than or equal to the ones of policy π' for all states-action pairs: $Q^{\pi}(s, a) \geq Q^{\pi'}(s, a)\ \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$. The best policy is denoted π^* and its matching value function $Q^{\pi^*}(s, a)$ or $Q^*(s, a)$ is called *optimal value function*. It is the goal of value-based RL to find Q^* and π^*, or a close approximation of them.

5.3.1 Discrete Value-Based RL

In discrete value-based RL, both states and actions are chosen from a finite, discrete set. Thus, the action-value function introduced in Eqn. (5.3) can be expressed as a table. In addition, with a finite number of states and actions, there is always a unique solution for Q^{π} but it may not be tractable to calculate the optimal value function. Sutton and Barto (1998) is an excellent resource for the different methods of approximating the optimal value function. Below, some of the more important algorithms are summarised.

Monte Carlo Policy Iteration

One way of finding a (close to) optimal value function and policy is called the *Monte Carlo policy iteration* method, which is an approximation to model-based policy iteration (see Chapter 5.2). It starts with a random policy π_0, executes a step of *policy evaluation* to get the matching value function Q^{π_0}, followed by a step of *policy improvement*, that generates a better policy π_1 from the current value function, and so on. Eventually, the cycle will produce a policy close to the optimal policy π^*. Figure 10 illustrates the cycle of policy iteration.

The policy evaluation step can be executed by collecting real experiences from following policy π_k and averaging over all returns R whose histories included a certain state-action pair: $Q^{\pi_k}(s,a) = \langle R^h \rangle$ for all histories h containing (s,a). This is also called *Monte Carlo* policy evaluation. The policy improvement step then uses the approximated value function Q^{π_k} to create a policy π_{k+1} that is at least as good or better than the original policy π_k. This is done by defining a policy that chooses the actions with maximum Q-value, also called a *greedy policy*: $\pi_{k+1}(s) = \arg\max_a Q^{\pi_k}(s,a)$. Eventually, this cycle converges to the optimal policy π^*.

Temporal Difference Learning

One evident draw-back of policy iteration is the fact that one has to wait for the episodes to finish to receive the returns. Even worse, in infinite-horizon settings the episode never finishes and thus a return can not be established. *Temporal Difference* algorithms use what is called *bootstrapping* to approximate the returns while the current episode is still ongoing. Bootstrapping basically uses a preliminary result of some calculation within the same calculation. In this case, the returns (see Eqn. (5.1) with $a_\Sigma = 1, a_D = \gamma^k$) required to calculate the Q-values are approximated by the Q-values, which leads to this recursive definition, called the *Bellman equation* for Q^π:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{R_t \mid s_t = s, a_t = a\} \\ &= E_\pi\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k r_{t+k+2} \mid s_t = s, a_t = a\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a\right\}, \end{aligned} \tag{5.3}$$

Eqn. (5.3) shows that in expectation, R_t can be approximated by $r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1})$. Now, in order to update the estimate for $Q^\pi(s,a)$, only the next interaction step is required instead of the whole history.

A further disadvantage of policy iteration was the growing amount of memory required to store all the returns in order to calculate the sample mean. This problem can be avoided by using an exponential moving average:

$$Q^\pi(s_t, a_t) \leftarrow (1-\alpha) Q^\pi(s_t, a_t) + \alpha \langle R^h \rangle \tag{5.4}$$

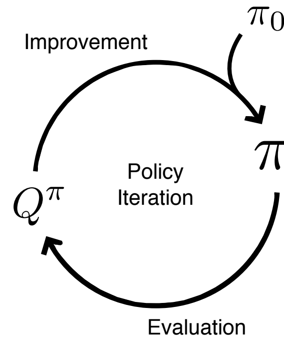


Figure 10: The policy iteration cycle. Starting from an initial (random) policy π_0 it first applies a policy evaluation step, which produces a value function Q^π from the current policy π followed by a policy improvement step, which generates a new policy π' from the value function Q^π that is better or at least equal to π.

Combined with the results from eqn. (5.3), this leads to the well-known *SARSA* algorithm (Rummery and Niranjan, 1994; Sutton, 1996):

$$Q^\pi(s_t, a_t) \leftarrow (1-\alpha)Q^\pi(s_t, a_t) + \alpha\Big(r_{t+1} + \gamma Q^\pi\,(s_{t+1}, a_{t+1})\Big) \tag{5.5}$$

The name SARSA derives from the necessary "ingredients" to execute one update step according to (5.5), namely the state, action, reward, next state and next action. SARSA is an *on-policy* algorithm, because it updates the Q-values while following the current policy π.

Probably even more prominent is a variation of the SARSA algorithm, called *Q-Learning* (Watkins and Dayan, 1992). In Q-Learning, the term $Q^\pi(s_{t+1}, a_{t+1})$ in (5.5) is replaced by $\max_a Q^\pi(s_{t+1}, a)$. Its update rule—commonly presented in a slighly re-arranged form—then becomes:

$$Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha\Big(r_{t+1} + \gamma \max_a Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t)\Big) \tag{5.6}$$

This small change transformed SARSA into an *off-policy* algorithm that assumes a greedy policy for the calculation of the next-step Q-value. It can update the value function for policy π while following another policy. This is important for example if the environment bears certain risks that policy π has not learned to master yet. A more safety-oriented policy can be used to generate the interaction samples instead.

Both SARSA and Q-Learning can be executed *online* (which is not to be confused with the term on-policy). The agent interacts with the world and updates the value function after each interaction step. The policy can be updated on the fly as well, in the same way as before, by using a greedy policy for action selection:

$$\pi(s) = \arg\max_a Q^\pi(s, a) \tag{5.7}$$

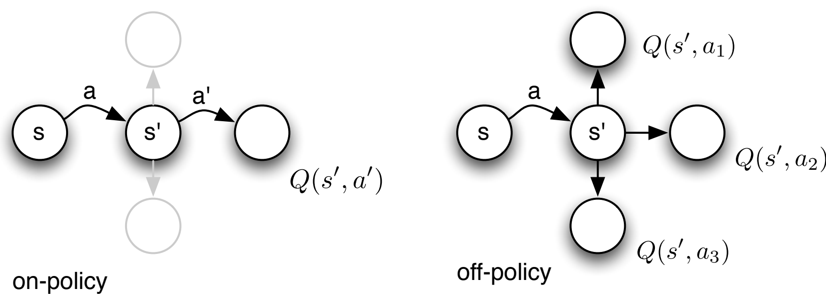


Figure 11: Difference between on-policy (left) and off-policy (right). On-policy methods use the executed action a' while following the policy. Off-policy methods use the maximum of all available actions according to the current value function. They can update a policy while following another.

However, the value function is initialised with random values and only slowly converges while executing the learning loop. This means that the actions chosen by a greedy policy can be poor early on. To avoid getting stuck or running into dead ends repeatedly, some exploration mechanism is usually included into the policy. One way of making sure to visit all states (roughly) equal amounts of time is the idea of *exploring starts*. The environment is initialised with a randomly selected state for each episode. This of course only works with environments that allow starting from random states, e.g. in simulations. Other forms of exploration, suitable for real-world environments that do not allow exploring starts, will be discussed in section 5.5.

5.3.2 Continuous Value-Based RL

Applying discrete Reinforcement Learning algorithms to real-world problems quickly reaches its limits and the number of states and actions become too many to keep the problem tractable. Furthermore, in many domains, including robotics and computer vision, one often deals with strictly continuous domains in the first place. Finally, generalisation between similar states—something that discrete methods are unable to do—can be of great advantage to speed up the learning process. This is where continuous RL methods shine. Continuous value-based methods come in a great variety. First, we need to differentiate between the case of continuous states with discrete actions and both continuous states and actions. In what follows, we will call the former *continuous state* case and the latter *continuous action* case, which will always include continuous states as well. The combination of discrete states with continuous actions is usually not covered separately and will be treated together with the continuous-action case.

The main issue with a continuous control signal is that action selection becomes much harder, because the term $\arg\max_a Q^{\pi}(s,a)$ from Eqn. (5.7) is not easily computable. With a finite number of actions, it was possible to compare all Q-values containing action a against each other (simply by calculating $Q^{\pi}(s,a_i)$ for all $a_i \in \mathcal{A}$) and to pick the highest one. With infinitely many actions, comparing all Q-values is not possible.

Using a general function approximator (FA) to estimate the Q-values for state-action pairs, it is possible but expensive to follow the gradient $\frac{\partial Q(s,a)}{\partial a}$ towards an action that returns a higher Q-value. In actor-critic architectures (Sutton and Barto, 1998), where the policy (the actor) is separated from the learning component (the critic), one can back-propagate the temporal difference error through the critic FA (usually implemented as neural networks) to the actor FA and train the actor to output actions that return higher Q-values (Riedmiller, 2005; van Hasselt and Wiering, 2007).

There are some alternatives, though, that will be mentioned further below. For now, we will deal with discrete actions and tackle the problem of continuous states.

Value-Based RL for Continuous States

One solution (which is more of a work-around) for dealing with a continuous state space is to separate the states into discrete clusters and use conventional discrete RL to solve the problem. If detailed information about the structure of the state space is available, the separation of states can be done manually. *Tile Coding*, for example, splits the state space into disjunct tiles spread out to cover the complete state space. Each continuous state covered by one tile is then assigned a single value and treated as one discrete state. *Sparse Coarse Coding* is another technique of discretisation (Sutton, 1996). If the distribution of states and their limits is unknown, however, the placement of tiles is not trivial. Clustering methods can help to find discrete state clusters. They are more flexible in a way that they can cover large, only rarely visited areas of the state space with only one or a few clusters, while putting the majority of clusters in areas of dense and frequent states. If the given problem is benign and small variations in states do not require drastically different actions, this approach can work well. But often, especially with high-dimensional problems, this is not the case and clustering the state space will fail, due to the curse of dimensionality.

As an alternative, function approximation methods, as introduced in Section 4, can be used to interpolate over the state space. The straightforward approach would be to observe the transition from state s to s'

with action a and reward r and train the function approximator with the one-step look-ahead target $r + \gamma \max_a Q(s', a)$ to move the outputs closer to the optimal value function Q^*, which would be the continuous extension of the Q-Learning algorithm. One would expect this to converge just like in the tabular case. But the convergence proofs that guarantee global convergence of the value functions to the optimum only hold for the discrete case, where the exact values can be stored in a table. And while there are some positive examples (Samuel, 1969; Tesauro, 1994; Antos et al., 2008), where function approximation was successfully used in Reinforcement Learning, it has been shown that even for simple toy problems, the value function approximations can become unstable and even diverge (Boyan and Moore, 1995; Baird, 1995). One possible reason is a systematic over-estimation of the utility values by the function approximator (Thrun and Schwartz, 1993). Despite the lack of convergence proof and numerous counter-examples, a number of continuous value-based methods, together with some techniques to avoid divergence, have been suggested.

Residual Gradient Methods (Baird, 1995) offer a solution to the issue of divergence, by using a different target, derived from the *mean squared Bellman residual* $E = \frac{1}{n} \sum_s \left(\langle R + \gamma \max Q(s', a) \rangle - Q(s, a) \right)^2$. The updates for the parameters of the function approximator would then be:

$$\Delta\theta = \alpha \left(r + \gamma \max_a Q(s', a) - Q(s, a) \right) \left(\gamma \frac{\partial}{\partial\theta} \max_a Q(s', a) - \frac{\partial}{\partial\theta} Q(s, a) \right)$$

While Baird could prove the convergence to the optimal value function Q^*, he also pointed out that this class of algorithms might converge very slowly. He then suggested a hybrid version, which he called *Residual Algorithms*, that use batch training on a weighted average of the direct update and the residual gradient to speed up the learning process while at the same time not violating the convergence criteria.

An important aspect for successful continuous value-based training appears to be the batch update of the Q-function, which is also one key element in *Fitted Q-Iteration* (Ernst et al., 2005). For this class of algorithms the value function can be represented by any regression method (including non-parametric ones), and it is trained by collected experience, stored as 4-tuples (s, a, r, s'), i.e. state, action, reward and the next state. For the regression task, a dataset is prepared that consists of (s, a) as input and $(r + \gamma \max_a Q(s', a))$ as target for each collected 4-tuple. The Q-values for the targets come from the function approximator itself (bootstrapping, just like the discrete case). After training the regression method until convergence, the procedure is repeated and a a new dataset is created in the same way. Again, the outputs of the previous function approximator are used as the Q-values for the targets to train

the current one. This procedure is repeated until a satisfactory level of performance is achieved.

Various types of non-linear function approximators have been successfully used with FQI, e.g. Neural Networks (Riedmiller, 2005; Hafner, 2009; Hafner and Riedmiller, 2011), CMACs (Timmer and Riedmiller, 2007), Gaussian Processes (Rasmussen and Williams, 2006; Deisenroth et al., 2009), Advantage Weighted Regression (Neumann and Peters, 2009), and many others (Neumann et al., 2006). To avoid divergence, many of these approaches use *experience replay*, where all the collected transition tuples are constantly presented to the model again.

Value-Based RL for Continuous Actions

In a continuous action space, action selection is much harder, mainly because calculating $\arg\max_a Q(s,a)$ is not trivial. With discrete (and more importantly finitely many) actions, action values for all actions in a given state s can be calculated and compared to each other to pick the highest one. With continuous (and therefore infinitely many) actions, another method of finding the action with the maximum value is required. Corroborating this matter, Gaskett et al. (1999) present eight properties that they believe to be necessary for value-based continuous action RL, one of which is called "Action Selection: find action with the highest expected value quickly". They use a special type of value function approximator called *wire fitting* (Baird and Klopf, 1993). Wire-fitting is a function approximation method designed to return the maximum of all output values in constant time. This is done by interpolating a function between a few given control points (or in higher dimensions control wires), where the update rule for adding new samples will always ensure that one of the control points is located at the maximum value. Finding the maximum is then merely a constant operation of going through the set of control points.

Another technique of finding the maximum value is called Gradient Ascent on the Value (GAV). This method uses an actor $\pi(s) = a$, mapping states to actions, and a critic $Q(s,a) = q$ that maps state and action to a value (Prokhorov et al., 2002; Hafner, 2009). The changes to the parameters θ_q of the critic can be calculated by back-propagating the temporal difference error through the approximator as usual:

$$\Delta\theta_q = \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big)\frac{\partial Q(s_t, a_t)}{\partial \theta_q} \tag{5.8}$$

Regarding the update of the actor, the parameters θ_π have to be changed towards a higher Q-value, so the gradient information from the critic

can be back-propagated down to the parameters of the actor, followed by gradient *ascent* to increase the overall value:

$$\Delta\theta_\pi = \alpha \frac{\partial Q(s_t, a_t)}{\partial A(s_t)} \frac{\partial A(s_t)}{\partial \theta_\pi} \tag{5.9}$$

One problem is here, that while the critic is not fully trained, the gradient information will not be accurate yet, and the actor parameters might be pushed in the wrong direction. significant improvement was observed when the probability of updating the actor slowly increased during experiments (van Hasselt and Wiering, 2007).

Hafner applied GAV to a neural network architecture in his PhD thesis (Hafner, 2009), and called it Neural Fitted Q-Iteration with Continuous Actions (NFQCA). Actor and critic are represented as neural networks, entangled in a triangular architecture like shown in Figure 12. One update step consists of forward-propagating the state through the actor network to get an action, and then state and action together through the critic network for a value. The temporal difference error is then back-propagated, first through the critic network and then further back through the actor.

Continuous Actor Critic Learning Automaton (CACLA), another actor-critic architecture, only uses the sign of the gradient for updates of the actor, and only positive improvements for updates to both actor and critic, i.e. if the temporal difference error is positive (van Hasselt and Wiering, 2007). Comparing it to wire-fitting and GAV, the authors receive overall better results with CACLA on common benchmarks.

A method has been proposed to use an incremental topology preserving map (ITPM), using units with adaptive resolution to cover the space of continuous actions (Millán et al., 2002). These units are added incrementally, memorize a number of Q-values for certain areas of each unit's responsible action space area, and the resulting continuous action is an average weighted by the Q-values over the discrete actions of a unit.

Another interesting, yet quite different way of getting continuous actions is called Binary Action Search (Pazis and Lagoudakis, 2009). Here, instead of retrieving an action for a given state directly, a binary policy instead answers the question whether the current action should be increased or decreased. A second key aspect of the idea is that the policy is queried several times for one action, basically implementing a binary search in action space. The state is augmented with the current action, and the binary search starts at $(a_{\max} - a_{\min})/2$. If the value for the action *increase* is higher, the next query will be for the action in the middle of

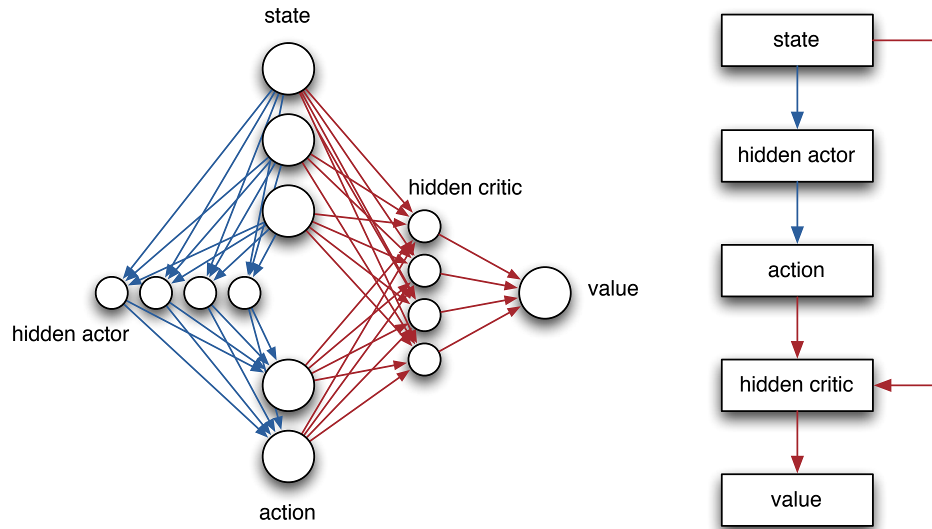


Figure 12: NFQCA network architecture. Left: The blue connections represent the actor network, mapping states to actions. The red connections show the critic network, mapping from state and action to a single value. Back-propagation works in reverse order: the temporal difference error is propagated back through the critic network to the action and then further back through the actor network. Right: Simplified network architecture.

$(a_{\max}-a_{\min})/2$ and $a_{\max}$, and if the action value for *decrease* is higher, the next query point will be between $a_{\min}$ and $(a_{\max}-a_{\min})/2$. This procedure is repeated several times, each time halving the search interval, until the desired accuracy of the action is achieved. Most continuous-state Reinforcement Algorithms can be equipped with binary action search to deliver continuous actions, the only requirements being that they use continuous states and are able to produce a binary decision.

To summarise, it seems that the main issue with value-based RL with continuous actions lies in selecting the action with maximum value, as an exhaustive comparison of all actions is impossible. Several approaches to find $\arg\max_a Q(s,a)$ have been suggested before, of which a few were presented here: custom function approximators specifically designed for fast access to the maximum action, gradient descent through the critic approximator down to the actor and binary search in action space. A different approach on continuous actions, albeit without the need for value functions, will be presented in the next chapter “Direct Reinforcement Learning”.

5.4 DIRECT REINFORCEMENT LEARNING

Direct reinforcement learning methods, in particular Policy Gradient methods (Williams, 1992; Peters and Schaal, 2006, 2008a,b), avoid the problem of finding the action with maximum value $\arg\max_a Q(s,a)$ altogether, thus being popular for continuous action and state domains. Instead, states are mapped to actions directly by means of a parameterised function approximator, without utilising Q-values (basically, this is equivalent to the actor-critic architecture without a critic). The parameters θ are changed by following a gradient representing the performance of a given policy (Baird and Moore, 1999). Different approaches exist to estimate this gradient (Peters and Schaal, 2008a).

5.4.1 Overall performance measure

First, an overall performance measure $J(\pi)$ is defined for a given policy π, independent of any history. It represents the expected return over all histories h^π that are possible when following policy π:

$$\begin{aligned} J(\pi) &= E\{R(h^\pi)\} \\ &= \int p(h^\pi)R(h^\pi)\, dh^\pi \end{aligned} \tag{5.10}$$

In order to optimise policy $\pi(s;\theta)$, the parameters θ are moved along the gradient of J towards an optimum. The gradient $\nabla_\theta J(\pi)$ is defined as

$$\begin{aligned} \nabla_\theta J(\pi) &= \nabla_\theta \int p(h^\pi)R(h^\pi)\, dh^\pi \\ &= \int \nabla_\theta p(h^\pi)R(h^\pi)\, dh^\pi \end{aligned} \tag{5.11}$$

Instead of $J(\pi)$ for the performance of policy π parameterised with parameters θ, we also write $J(\theta)$ directly.

In order to calculate (or approximate) this gradient, two general directions are possible: *finite difference* methods and *likelihood ratio* methods.

5.4.2 Finite Difference Methods

One way to calculate the gradient estimate are finite difference methods, which perturb the policy parameterised by θ by some small amount $\delta\theta$ and use the difference in performance to approximate the gradient. The

new policy is compared to a reference point J_{ref}, which can be equal to $J(\theta)$ for forward difference methods or $J(\theta - \delta\theta)$ for central difference methods. The gradient is then approximated by the difference quotient:

$$\nabla_\theta J(\theta) \approx \frac{J(\theta + \delta\theta) - J_{\text{ref}}}{\delta\theta} \tag{5.12}$$

In order to get a more exact approximation, several parameter perturbations are usually collected (one for each roll-out) and the gradient is then estimated through linear regression. For this, several roll-outs are generated by adding some exploratory noise to the policy parameters, resulting in actions $a = f(s; \theta + \delta\theta)$. From the roll-outs, the matrix Θ is formed, which has one row for each parameter perturbation $\delta\theta_i$. The column vector J contains the corresponding performances $J(\theta + \delta\theta)$ in each row:

$$\Theta_i = [\, \delta\theta_i \quad 1 \,] \tag{5.13}$$

$$J_i = [\, J_i(\theta + \delta\theta) \,] \tag{5.14}$$

The ones in the right column of Θ are needed for the bias in the linear regression. With forward differences, the J_{ref} from Eqn. (5.12) cancel out and are are not present anymore in J. By least squares linear regression, the gradient can now be estimated with the Moore-Penrose pseudo-inverse:

$$\beta = (\Theta^T \Theta)^{-1} \Theta^T J, \tag{5.15}$$

where the first n elements of β are the n components of the gradient $\nabla_\theta J(\theta)$, one for each dimension of θ.

5.4.3 Likelihood Ratio Methods

Rather than perturbing the policy directly, which usually requires a fair amount of system knowledge, likelihood ratio methods perturb the resulting action instead, leading to a stochastic policy such as

$$a = f(s; \theta) + \epsilon, \; \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{5.16}$$

In this case, the exploratory noise ϵ is normally distributed around zero with a certain variance σ^2. Unlike with finite difference methods, the new policy that leads to the behaviour expressed in Eqn. (5.16) is no longer known and $J(\theta + \delta\theta)$ from Eqn. (5.12) and consequently the whole difference quotient can no longer be calculated. Thus, likelihood ratio methods use a different approach in estimating the gradient of $J(\pi)$ with respect to θ.

Derivation of the Likelihood Ratio Gradient

For the derivation of the likelihood ratio gradient, the Markov assumption is not needed. Therefore, the derivation is presented in its general form, where actions can depend on the whole history.

The probability of observing a history h^π under policy π is given by the probability of starting with an initial observation s_0, multiplied by the probability of taking action a_0 under h_0 (which equals s_0), multiplied by the probability of receiving the next observation s_1, and so on. Thus, Eqn. (5.17) gives the probability of encountering a certain history.

$$\begin{aligned} p(h^\pi) &= p(s_0)\pi(a_0|h_0^\pi)p(s_1|h_0^\pi, a_0)\pi(a_1|h_1^\pi)p(s_2|h_1^\pi, a_1)\dots \\ &= p(s_0)\prod_{t=1}^{T}\pi(a_{t-1}|h_{t-1}^\pi)\,p(s_t|h_{t-1}^\pi, a_{t-1}) \end{aligned} \tag{5.17}$$

Taking Eqn. (5.11) as a starting point, it can be rewritten by making use of the fact that $1/x \cdot \nabla x = \nabla \log(x)$, which is sometimes referred to as the *log trick* in the literature. By multiplying with $1 = p(h^\pi)/p(h^\pi)$ inside the integral we get

$$\nabla_\theta J(\pi) = \int \frac{p(h^\pi)}{p(h^\pi)} \nabla_\theta p(h^\pi) R(h^\pi)\, dh^\pi, \tag{5.18}$$

which yields after applying the log trick:

$$\nabla_\theta J(\pi) = \int p(h^\pi) \nabla_\theta \log p(h^\pi) R(h^\pi)\, dh^\pi \tag{5.19}$$

The next few steps only consider the gradient $\nabla_\theta \log p(h^\pi)$. Substituting the probability $p(h^\pi)$ according to Eqn. (5.17) gives

$$\nabla_\theta \log p(h^\pi) = \nabla_\theta \log \Big[p(s_0) \prod_{t=1}^{T} [\pi(a_{t-1}|h_{t-1}^\pi)\, p(s_t|h_{t-1}^\pi, a_{t-1})] \Big] \tag{5.20}$$

which, by the rules of the log, transforms into

$$\begin{aligned} \nabla_\theta \log p(h^\pi) = \nabla_\theta \Big[&\log p(s_0) + \sum_{t=1}^{T} \log \pi(a_{t-1}|h_{t-1}^\pi) + \\ &\sum_{t=1}^{T} \log p(s_t|h_{t-1}^\pi, a_{t-1}) \Big] \end{aligned} \tag{5.21}$$

On the right side of Eqn. (5.21), only the policy π is dependent on θ, so the gradient can be simplified to

$$\nabla_\theta \log p(h^\pi) = \sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t-1}|h_{t-1}^\pi) \tag{5.22}$$

We can now re-substitute this term into Eqn. (5.19) and get

$$\begin{aligned}\nabla_\theta J(\pi) &= \int p(h^\pi) \cdot \sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t-1}|h^\pi_{t-1}) \cdot R(h^\pi)\, dh^\pi \\ &= E\left\{\sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t-1}|h^\pi_{t-1}) \cdot R(h^\pi)\right\}\end{aligned} \tag{5.23}$$

Unfortunately, the probability distribution $p(h^\pi)$ over the histories produced by π is not known in general. Thus, we need to approximate the expectation, e.g. by *Monte-Carlo sampling*. Therefore, we collect N samples through world interaction, where a single sample comprises a complete history h^π (one episode or roll-out) to which a return $R(h^\pi)$ can be assigned and sum over all samples. This leads to the well-known REINFORCE (Williams, 1992) gradient estimate:

$$\nabla_\theta J(\pi) \approx \frac{1}{N} \sum_{h^\pi} \sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t-1}|h^\pi_{t-1}) \cdot R(h^\pi) \tag{5.24}$$

Variance Reduction Techniques

Because the integral $\int \nabla_\theta p(h^\pi)\, dh^\pi = \nabla_\theta 1 = 0$ always yields zero, a constant baseline in the policy gradient estimate does not bias the estimator but can reduce the variance significantly. When adding a baseline value b, Eqn. (5.23) becomes

$$\nabla_\theta J(\pi) = E\left\{\sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t-1}|h^\pi_{t-1})\, (R(h^\pi) - b)\right\} \tag{5.25}$$

For the baseline b a moving average of recent returns can be used, with the update rule $b \leftarrow (1-\alpha)b + \alpha(R_t)$ and $\alpha \in [0,1]$. However, an optimal baseline should minimise the variance of the gradient estimate as much as possible, and can be done explicitly (Peters and Schaal, 2008a). The baseline is given for each dimension d of the gradient as:

$$b^d = \frac{\left\langle \left(\sum_{t=1}^{T} \nabla_{\theta_d} \log \pi(a_{t-1}|h^\pi_{t-1})\right)^2 R(h^\pi) \right\rangle}{\left\langle \left(\sum_{t=1}^{T} \nabla_{\theta_d} \log \pi(a_{t-1}|h^\pi_{t-1})\right)^2 \right\rangle} \tag{5.26}$$

So far, the gradient estimate uses the full return as the sum over all rewards of an episode $R(h^\pi) = \sum_{t=1}^{T} r_t$ for an update (discounting ignored). Assuming that the policy does not change during an episode,

and important observation is that past rewards do not influence the current and future actions. This is called the Policy Gradient Theorem (Sutton et al., 2000) or G(PO)MDP (Baxter and Bartlett, 2001). Including this idea in REINFORCE, the gradient estimate can be rewritten as:

$$\nabla_\theta J(\pi) = E\left\{\sum_{t=1}^{T} \nabla_\theta \log \pi(a_{t-1}|h^\pi_{t-1}) \left(\sum_{k=t}^{T} r_t - b\right)\right\} \tag{5.27}$$

5.5 EXPLORATION

Exploration is a critical component of RL, affecting both the number of trials required and the quality of the solution found. Novel solutions can be found only through effective exploration. Preferably, exploration should be broad enough not to miss good solutions, economical enough not to require too many trials and intelligent in the sense that the information gained through it is high. Clearly, those objectives are difficult to trade off, a problem known as the *exploration / exploitation* dilemma: without exploration, the agent can only go for the best solution found so far, not learning about potentially better solutions. Too much exploration leads to mostly random behaviour without exploiting the learned knowledge. A good exploration strategy carefully balances exploration and greedy policy execution. Below we will look at a few common exploration techniques for discrete (section 5.5.1) and continuous (section 5.5.2) action spaces, explain the key differences of action-based vs. parameter-based exploration (section 6.2) and finally introduce a methodology for bridging the gap between these two approaches, called state-dependent exploration (section 6.3).

5.5.1 Exploration in Discrete Action Spaces

Many exploration techniques have been developed for the case of discrete actions (Sutton, 1990; Schmidhuber, 1991b; Thrun, 1992; Cohn et al., 1994; Wiering and Schmidhuber, 1998; Milano et al., 2001; Abbeel and Ng, 2005), commonly divided into undirected and directed exploration. The most popular—albeit not the most effective—undirected exploration method is ϵ-greedy exploration, where some of the actions are selected randomly, and the probability of choosing a random action decreases over time. In practice, a random number r is drawn from a uniform distribution $r \sim \mathcal{U}(0, 1)$, and action selection follows this rule:

$$\pi(s) = \begin{cases} \arg\max_a Q(s, a) & \text{if } r \geq \epsilon \\ \text{random action from } \mathcal{A}(s) & \text{if } r < \epsilon \end{cases}$$

where $\mathcal{A}(s)$ is the set of valid actions from state s and $0 \leq \epsilon \leq 1$ is the trade-off parameter, which is reduced over time to slowly transition from exploration to exploitation.

The problem with ϵ-greedy exploration is that the random actions could be really bad choices and would ruin an otherwise well-performing episode. Think of a balancing task, where a robot has to carefully balance a pole on its actuator, by moving the actuator in the horizontal plane. One little mishap would tip over the pole and the whole episode would be penalised with a negative reward.

One way to prevent such unwanted exploratory accidents is Boltzmann exploration. This exploration technique never follows the policy in a completely greedy fashion, but instead always chooses an action with a probability that is proportional to its value given by the current state s and the action value function Q.

$$p(a_t|s_t) = \frac{e^{Q(s_t,a_t)/\tau}}{\sum_a e^{Q(s_t,a)/\tau}}.$$

This means that the better an action is under the current policy, the more likely it is chosen. Very bad actions are much less likely to be explored under Boltzmann exploration, but it is still possible, particularly in the beginning. Just as with ϵ-greedy exploration, Boltzmann exploration has a temperature parameter τ that starts at an initially high value, so that all actions are equally likely to be chosen. Over time, τ is slowly reduced during learning for greedier selection towards the end.

Another interesting category of exploration mechanisms is that of Artificial Curiosity (Schmidhuber, 1991a, 1999, 2007; Graziano et al., 2011), or Intrinsical Reinforcement Learning (Singh et al., 2005), where (at least a part of) the policy goal is to explore the yet unknown areas of the state space to gain knowledge and increase predictability of future states.

In the next section we will concentrate on continuous actions.

5.5.2 Exploration in Continuous Action Spaces

In the case of continuous actions, exploration is often neglected. If the policy is considered to be stochastic, a Gaussian distribution of actions is usually assumed, where the mean is often selected and interpreted as the greedy action:

$$a \sim \mathcal{N}(a_{\text{greedy}}, \sigma^2) = a_{\text{greedy}} + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{5.28}$$

During learning, exploration then occurs implicitly — almost as a side-effect — by sampling actions from the stochastic policy. While this is

convenient, it conceals the fact that two different stochastic elements are involved here: the exploration and the stochastic policy itself. This becomes most apparent if we let σ adapt over time as well, following the gradient $\frac{\partial J(\theta)}{\partial \sigma}$, which is well-defined if the policy is differentiable. If the best policy is in fact a deterministic one, σ will decrease quickly and therefore exploration comes to a halt as well. This clearly undesirable behaviour can be circumvented by adapting the variance manually, e.g. by decreasing it slowly over time.

Another disadvantage of this implicit additive exploration is the independence of samples over time. In each time step, we draw a new ϵ and add it to the actions, leading to a very noisy trajectory through action space (see Figure 14 top). A robot controlled by such actions would exhibit a very shaky behaviour, with a severe impact on the performance. Imagine an algorithm with this kind of exploration controlling the torques of a robot end-effector directly. Obviously, the trembling movement of the end-effector will worsen the performance in almost any object manipulation task. And we ignore the fact that such consecutive contradicting motor commands might even damage the robot or simply cannot be executed. Thus applying such methods requires the use of motion primitives (Schaal et al., 2004) or other transformations. Despite these problems, many current algorithms (Williams, 1992; Riedmiller et al., 2007; Peters and Schaal, 2008b) use this kind of Gaussian, additive action-perturbing exploration, see Eqn. (5.28). Alternatives, that avoid the above problems by sampling in parameter space rather than action space, are presented in Chapter 6.

Part III

Methods and Experiments

6 STATE-DEPENDENT EXPLORATION

6.1 INTRODUCTION

As discussed in chapter 5.5, exploration is indispensable in order to successfully learn through reinforcement. To put it in a nutshell: Without trying out new behaviour, the agent would be unable to improve. Most real-world examples are continuous in nature, and exploration for continuous action spaces is often limited to adding some form of random noise (usually from a Gaussian distribution centred at the deterministic action value) to the action. This kind of exploration is unsubstantiated and random at best, and can have severe negative impact on learning performance at worst. The amount of exploration is usually chosen arbitrarily with a decay factor decreasing the variance of the normal distribution over time. This approach requires a fair amount of system knowledge to fine-tune the amount of exploration and to avoid that the exploratory part of the final action dominates the actual deterministic part (i.e., the learned part). Even worse, in situations where actions have a temporal dependency, for example in robot control tasks, adding random noise to each consecutive step of the controller signal results in jitter and can not only cause poor performance but also damage to the hardware. Adding this kind of uninformed random noise to the actions also causes an increase in variance and makes credit assignment very difficult.

Driving to Work – An Example

Imagine you moved to another, unfamiliar city for a new job. Each morning, you take your car to work and try to find the shortest way to the office. On the first day (Monday), it took you 25 minutes to get there but you think you can do better. For the sake of this example, let's assume you don't have access to any maps but must find the way simply by driving there. How would you improve your daily route to the office to get there quicker?

If on Tuesday you take the same way as on Monday, you would get there in 25 minutes (neglecting any change in traffic). Obviously you're not off worse than yesterday, but you also didn't improve. This is called *exploitation* in Reinforcement Learning jargon, and it means that you used your

previously acquired knowledge to get an acceptable result. However, in order to improve in the future, you need to do the opposite: *exploration*. So on Tuesday, you decide to drive another way. The equivalent of exploring by adding random noise to actions, is to choose a random action each time you're forced to make a decision. So at each intersection, you draw a random number (by rolling dice, flipping a coin or some other means that gives you a uniform number between 1 and the number of streets to choose from) and follow the street determined by the drawn number. Does that procedure get you to work? Most likely not, because your actions have a temporal dependency. The action you took at the first intersection influences the choices you have at the next intersection. Many Reinforcement Learning problems are of such nature, and adding random, independent noise to each action is not helpful in such cases. But there is a second caveat: Let assume that you did in fact reach your office and even improved and made it in 22 minutes. In order to learn from this experience, you need to somehow associate the positive outcome with the actions you took. In Reinforcement Learning language, this is called "credit assignment". The problem here is though, that any of the choices you made that differed from the last trial could have caused this effect. Even worse, maybe some of the choices you took would have lead to a worse outcome, but were cancelled out by some really good choices. Here, the second problem of random exploration shows: It is very difficult to assign the credit back to actions which were overlaid with independent random noise. One solution is, to keep a list of all the choices ever made and store the return of every single episode (here: the time for one trip to work), and average over the results. So in our case, we would append "22 min" to each of the streets we took on Tuesday. If we drive often enough, we might be able to see some trends in the averages for each street, but it would take many episodes before they stand out from the noise.

In the following sections, we will describe an alternative to adding independent noise to actions. One method has already been introduced in Section 5.4.2: Finite differences explore by adding noise to the parameters of the controller, rather than the resulting action. We will talk more about parameter-exploring algorithms in Section 6.2 and discuss advantages and disadvantages, and then introduce a novel method that behaves like parameter exploration but is in fact manipulating actions. Why this is desirable and how it can be achieved will be subject of Section 6.3.

6.2 EXPLORATION IN PARAMETER SPACE

A significant problem with policy gradient algorithms such as REINFORCE (Williams, 1992), as described in section 5.4.3, is that the high variance in the gradient estimation leads to slow convergence. In the "Driving to Work" example, this equated to having to drive to work many times before we would be able to notice a trend in the different street return averages.

Various approaches have been proposed to reduce this variance (Baxter and Bartlett, 2000; Sutton et al., 2000; Aberdeen, 2003; Peters and Schaal, 2006). However, none of these methods address the underlying cause of the high variance, which is that repeatedly sampling from a probabilistic policy has the effect of injecting noise into the gradient estimate at every time step. Furthermore, the variance increases linearly with the length of the history (Munos, 2006), since each state may depend on the entire sequence of previous samples. An alternative to the action-perturbing exploration described (see section 5.5.2), is to alter the parameters θ of the policy directly.

Instead of manipulating the resulting actions as we saw with additive Gaussian exploration, parameter exploration adds a small perturbation $\delta\theta$ directly to the parameter θ of the policy before each episode, and follow the resulting policy throughout the whole episode. An example of such a parameter-exploring algorithm is the Finite Differences method from section 5.4.2.

Another interesting alternative is Policy Gradients with Parameter-Based Exploration (PGPE) (Sehnke et al., 2008a), where a distribution over the parameters of a controller is maintained and updated. Therefore PGPE explores purely in parameter space. The parameters are sampled from this distribution at the start of each sequence, and thereafter the controller is deterministic. Since the reward for each sequence depends only on a single sample, the gradient estimates are significantly less noisy, even in stochastic environments.

PGPE addresses the variance problem by replacing the probabilistic policy with a probability distribution over the parameters θ, i.e.

$$p(a_t|s_t,\rho) = \int_\Theta p(\theta|\rho)\delta_{F_\theta(s_t),a_t} d\theta, \tag{6.1}$$

where ρ are the parameters determining the distribution over θ, $F_\theta(s_t)$ is the (deterministic) action chosen by the model with parameters θ in state s_t, and δ is the Dirac delta function. The advantage of this approach is that the actions are deterministic, and an entire history can therefore be generated from a single parameter sample. This reduction in samples-per-history is what reduces the variance in the gradient estimate. As an

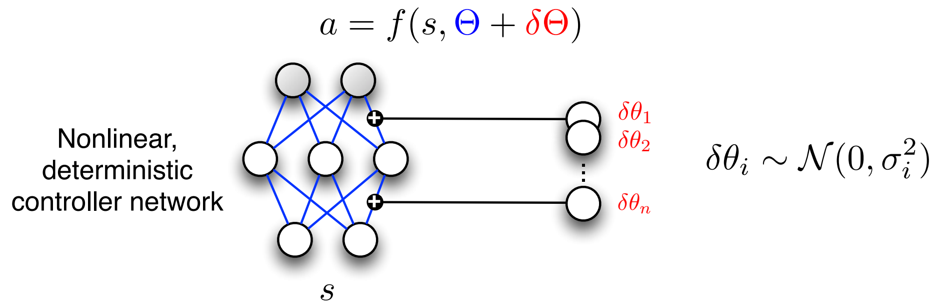


Figure 13: Parameter Exploration. Both PGPE and Finite Differences perturb the parameters of the controller directly, but differ in their approach to estimate the return gradient with respect to the controller parameters. Finite Differences sometimes uses a uniform rather than a normal distribution to sample the deltas, but the outcome is similar.

added benefit the parameter gradient is estimated by direct parameter perturbations, without having to back-propagate any derivatives, which allows the use of non-differentiable controllers.

Parameter-based exploration can have several advantages. First, we no longer need to calculate the derivative of the policy with respect to its parameters, since we already know which choice of parameters has caused the changes in behaviour. Therefore policies are no longer required to be differentiable, which in turn provides more flexibility in choosing a suitable policy for the task at hand. Second, when exploring in parameter space, the resulting actions come from an instance of the same family of functions. This contrasts with action-perturbing exploration, which might result in actions that the underlying function could never have delivered itself. In the latter case, gradient descent could continue to change the parameters into a certain direction without improving the overall behaviour. For example, in a neural network with sigmoid outputs, the exploratory noise could push the action values above $+1.0$. Third, parameter exploration avoids noisy trajectories that are due to adding i.i.d. noise in each time step. This fact is illustrated in Figure 14. Each episode is executed entirely with the same parameters, which are only altered between episodes, resulting in much smoother action trajectories. Furthermore, this introduces much less variance in the rollouts, which facilitates credit assignment and generally leads to faster convergence (Rückstieß et al., 2008a; Sehnke et al., 2008a).

But perturbing parameters instead of actions is not always possible or desired. With no system knowledge, changing parameters directly can lead to undefined behaviour of the controller, especially if it is non-linear. Small changes to the parameters of a non-linear system can lead to very big changes in the outcome. This could result in actions outside of the

safe working space of the system, or actions that simply cannot be executed by the system due to hardware constraints.

Another reason to avoid parameter perturbation is to optimize, or fine-tune, an already existing controller. The space of actions is likely flat and smooth around a good solution: A robot hand almost able to grasp an object will probably find a better solution by making small local changes to the end effector's trajectory. There is no guarantee, however, that the space of (non- linear) controller parameters is similarly smooth and flat around a solution. Small modifications could potentially be very destructive to an existing sub- optimal solution. This issue is particularly relevant for the robotics domain. Many manipulation tasks benefit from an initial teaching signal, for example a human leading an impedance-controlled robot (Urbanek et al., 2004) to seed the search for the controller parameters.

The next section introduces State-Dependent Exploration, a method that was developed to bring most of the benefits of parameter exploration to tasks where pure parameter perturbing methods, like PGPE or genetic algorithms are not an option.

6.3 STATE-DEPENDENT EXPLORATION

Now we will look at an alternative to parameter-based exploration, that addresses most of the shortcomings of action-based exploration. State-Dependent Exploration (SDE) (Rückstieß et al., 2008a) is compatible with standard policy gradient methods like REINFORCE in a way that it can simply replace or augment the existing Gaussian exploration described in Section 5.5. Since it is technically an additive action-based exploration method, it also circumvents the above mentioned problem of perturbing parameters in non-linear controllers. SDE, therefore, sits in between parameter-based and action-based exploration and inherits some of the positive aspects of both methods. It can be seen as an adapter for Likelihood Ratio Reinforcement Learning to behave more like parameter-based exploration, without actually modifying the parameters.

First, we will go beyond the introduction of REINFORCE from Section 5.4.3 by deriving the general case for multi-dimensional differentiable function approximators. Then we will modify REINFORCE's exploration technique and turn the algorithm into what we call State-Dependent Exploration.

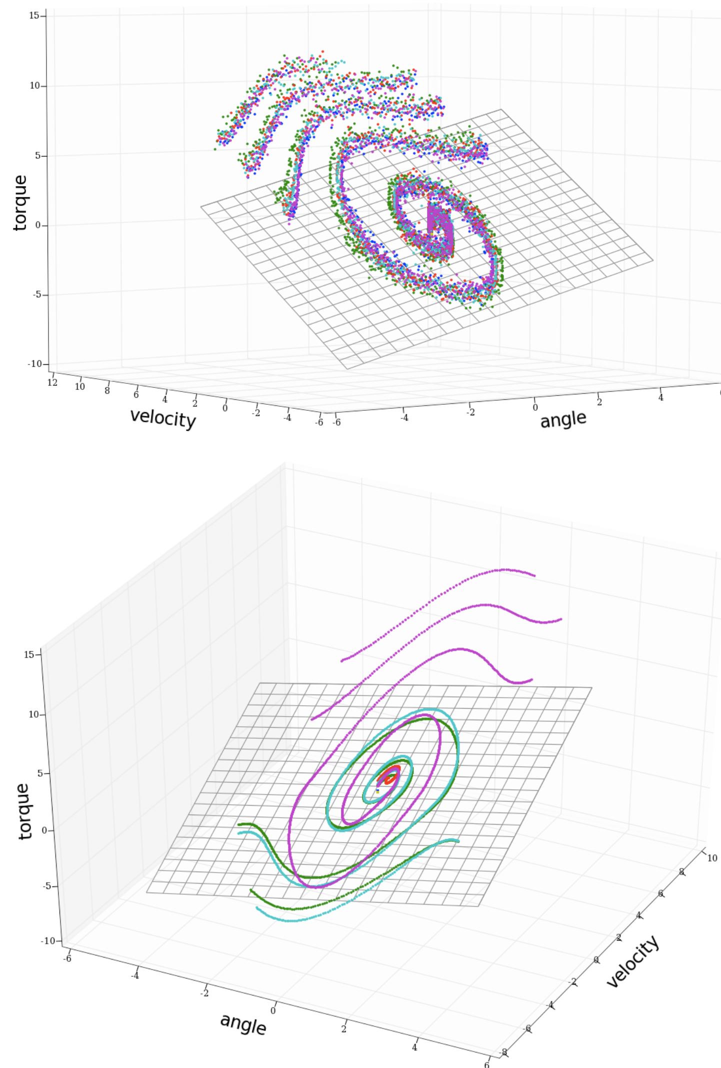


Figure 14: Illustration of the main difference between action (top) and parameter (bottom) exploration. Several roll-outs in state-action space of a task with state $x \in \mathbb{R}^2$ (velocity and angle axes) and action $a \in \mathbb{R}$ (torque axis) are plotted. While exploration based on action perturbation follows the same trajectory over and over again (with added noise), parameter exploration instead tries different *strategies* and can quickly find solutions that would take a long time to discover otherwise.

6.3.1 REINFORCE for General Multi-Dimensional Function Approximation

Here we describe how the results above, in particular Eqn. (5.24) from page 44, can be applied to general parametric function approximation.

Because we are dealing with both multi-dimensional states $\boldsymbol{s}$ and actions $\boldsymbol{a}$, we will now use bold font for (column) vectors in the notation for clarification.

Furthermore, to avoid the issue of a growing history length and to simplify the equations, we will assume the world to be Markovian for the remainder of this chapter, i.e. the current action only depends on the last state encountered, so that $\pi(a_t|h_t^\pi) = \pi(a_t|s_t)$. But due to its general derivation, the idea of SDE is still applicable to non-Markovian environments.

The most general case would include a multi-variate normal distribution function with a covariance matrix $\boldsymbol{\Sigma}$, but this would square the number of parameters and required samples. Also, differentiating this distribution requires calculation of $\boldsymbol{\Sigma}^{-1}$, which is time-consuming. We will instead use a simplification here and add independent uni-variate normal noise to each element of the output vector separately. This corresponds to a covariance matrix $\boldsymbol{\Sigma} = \text{diag}(\sigma_1, \dots, \sigma_n)$.[1] The action $\boldsymbol{a}$ can thus be computed as

$$\boldsymbol{a} = f(\boldsymbol{s}, \boldsymbol{\theta}) + \boldsymbol{e} = \begin{bmatrix} f_1(\boldsymbol{s}, \boldsymbol{\theta}) \\ \vdots \\ f_n(\boldsymbol{s}, \boldsymbol{\theta}) \end{bmatrix} + \begin{bmatrix} e_1 \\ \vdots \\ e_n \end{bmatrix} \tag{6.2}$$

with $\boldsymbol{\theta} = [\theta_1, \theta_2, \dots]$ being the parameter vector and f_j the jth controller output element. The exploration values e_j are each drawn from a normal distribution $e_j \sim \mathcal{N}(0, \sigma_j^2)$. The policy $\boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s})$ is the probability of executing action $\boldsymbol{a}$ when in state $\boldsymbol{s}$. Because of the independence of the elements, it can be decomposed into $\boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s}) = \prod_{k \in \mathbb{O}} \pi_k(a_k|\boldsymbol{s})$ with $\mathbb{O}$ as the set of indices over all outputs, and therefore $\log \boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s}) = \sum_{k \in \mathbb{O}} \log \pi_k(a_k|\boldsymbol{s})$. The element-wise policy $\pi_k(a_k|\boldsymbol{s})$ is the probability of receiving value a_k as kth element of action vector $\boldsymbol{a}$ when encountering state $\boldsymbol{s}$ and is given by

$$\pi_k(a_k|\boldsymbol{s}) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{(a_k - \mu_k)^2}{2\sigma_k^2}\right), \tag{6.3}$$

where we substituted $\mu_k := f_k(\boldsymbol{s}, \boldsymbol{\theta})$. We differentiate with respect to the parameters θ_j and σ_j:

$$\begin{aligned} \frac{\partial \log \boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s})}{\partial \theta_j} &= \sum_{k \in \mathbb{O}} \frac{\partial \log \pi_k(a_k|\boldsymbol{s})}{\partial \mu_k} \frac{\partial \mu_k}{\partial \theta_j} \\ &= \sum_{k \in \mathbb{O}} \frac{(a_k - \mu_k)}{\sigma_k^2} \frac{\partial \mu_k}{\partial \theta_j} \end{aligned} \tag{6.4}$$

1 A further simplification would use $\boldsymbol{\Sigma} = \sigma \boldsymbol{I}$ with $\boldsymbol{I}$ being the unity matrix. This is advisable if the optimal solution for all parameters is expected to lay in similar value ranges.

$$\begin{aligned}\frac{\partial \log \boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s})}{\partial \sigma_j} &= \sum_{k \in O} \frac{\partial \log \pi_k(a_k|\boldsymbol{s})}{\partial \sigma_j} \\ &= \frac{(a_j - \mu_j)^2 - \sigma_j^2}{\sigma_j^3} \end{aligned} \tag{6.5}$$

For the linear case, where $\boldsymbol{f}(\boldsymbol{s}, \boldsymbol{\theta}) = \boldsymbol{\Theta s}$ with the parameter matrix $\boldsymbol{\Theta} = [\theta_{ji}]$ mapping states to actions, (6.4) becomes

$$\frac{\partial \log \boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s})}{\partial \theta_{ji}} = \frac{(a_j - \sum_i \theta_{ji} s_i)}{\sigma_j^2} s_i \tag{6.6}$$

An issue with nonlinear function approximation (NLFA) is a parameter dimensionality typically much higher than their output dimensionality, constituting a huge search space for FD methods. However, in combination with LR methods, they are interesting because LR methods only perturb the resulting outputs and not the parameters directly. Assuming the NLFA is differentiable with respect to its parameters, one can easily calculate the log likelihood values for each single parameter.

The factor $\frac{\partial \mu_k}{\partial \theta_j}$ in (6.4) describes the differentiation through the function approximator. It is convenient to use existing implementations, where instead of an error, the log likelihood derivative with respect to the mean, i.e. the first factor of the sum in (6.4), can be injected. The usual backward pass through the NLFA then results in the log likelihood derivatives for each parameter (Williams, 1992).

6.3.2 Derivation from REINFORCE to SDE

Looking back at Eqn. (5.28) on page 46, it is obvious that for every time step, Gaussian exploration adds noise to the action a in order to explore different behaviour in the current situation. This is somewhat inefficient as it adds a lot of variance to the gradient estimates. Especially when encountering the same state s a second time, we will end up doing something different from what we did when we observed s for the first time. Generally, the reinforcement signal only covers the episode in total, returning an overall reward (called the *return*) for the complete trajectory which does not take into account single rewards at each time step. Here, the credit assignment does not work for state s because two different, possibly opposite actions, have been carried out. The information, which one (if any) had an effect on the return value, is not accessible.

Reducing the variance of the exploratory noise ϵ in Eqn. (5.28) does not solve this problem because the agent has to act *differently* in order to learn something new. An alternative to adding random noise is to add

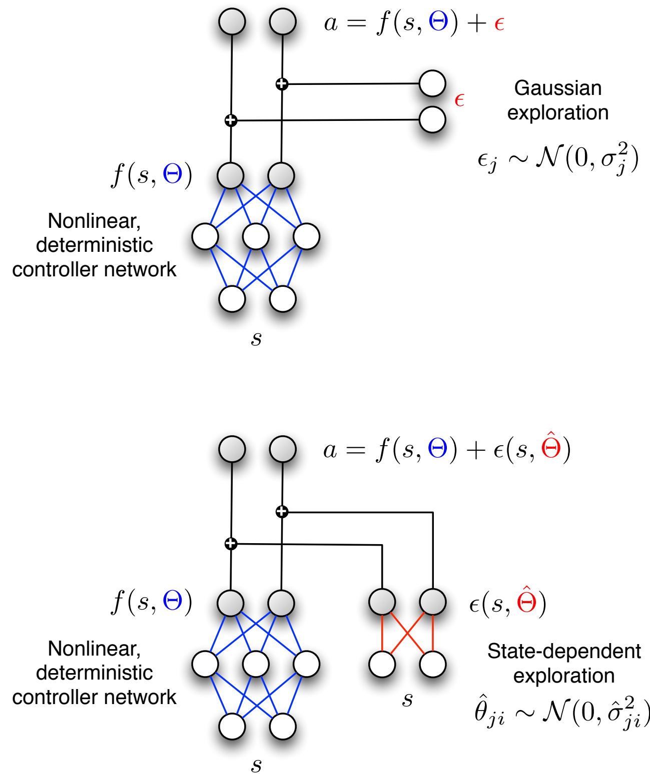


Figure 15: Difference between standard Gaussian exploration (top) and state-dependent exploration (bottom), explained graphically. Gaussian exploration adds a random value ϵ to the output of the deterministic controller at each time step. State-dependent exploration uses an exploration function $\epsilon(s, \hat{\Theta})$ that takes the state as input. Its output is added to the controller output. The parameters $\hat{\Theta}$ are drawn randomly. Without loss of generality, the controller is illustrated as a nonlinear feed-forward neural network and the exploration function as a linear mapping from the state. Other functions are possible as well.

state-dependent noise for exploration, which can still carry the necessary randomness but will always return the same value in the same state. This is achieved by a pseudo-random function $\epsilon(s; \hat{\Theta})$ that takes the state s as input. The randomness comes from the parameters $\hat{\theta}_{ji}$ which build the matrix $\hat{\Theta}$ and which are drawn from a normal distribution

$$\hat{\theta}_{ji} \sim \mathcal{N}(0, \hat{\sigma}_{ji}^2).$$

As illustrated visually in Fig. 15, bottom graphic, the action is calculated as follows, where f is the parameterised function approximator:

$$a = f(s, \Theta) + \epsilon(s, \hat{\Theta}), \quad \hat{\theta}_{ji} \sim \mathcal{N}(0, \hat{\sigma}_{ji}^2). \tag{6.7}$$

Instead of adding i.i.d. noise in each time step (cf. Equation 5.28), we introduce a pseudo-random function $\epsilon(s)$, that takes the current state as

Algorithm 1 State-Dependent Exploration (SDE)

1: **repeat**
2: **for** $n = 1$ to N **do**
3: draw random $\hat{\theta}_{ji} \sim \mathcal{N}(0, \hat{\sigma}_{ji}^2)$, $\hat{\Theta} = [\hat{\theta}_{ji}]$
4: **for** $t = 1$ to T **do**
5: observe state s_t^n
6: calculate $a_t^n \leftarrow f(s_t^n, \Theta) + \epsilon(s_t^n, \hat{\Theta})$
7: execute action a_t^n
8: receive reward r_t^n
9: store (s_t^n, a_t^n, r_t^n)
10: **end for**
11: **end for**
12: update π with collected samples $(s_t^n, a_t^n, r_t^n)\,\Big|_{t=1..T}^{n=1..N}$ according to Eqn. (5.24)
13: **until** convergence

input and is itself parameterised with parameters $\hat{\theta}$. These exploration parameters are in turn drawn from a Normal distribution with zero mean. The exploration parameters are varied between episodes (just like introduced in Section 6.2) and held constant during the roll-out. Therefore, the exploration function ϵ can still carry the necessary exploratory randomness through variation between episodes, but will always return the same value in the same state within an episode. Algorithm 1 shows a pseudo-code algorithm for state-dependent exploration.

Effectively, by drawing $\hat{\theta}$, we actually create a *policy delta*, similar to finite difference methods. In fact, if both $f(s;\Theta)$ with $\Theta = [\theta_{ji}]$ and $\epsilon(x, \hat{\Theta})$ with $\hat{\Theta} = [\hat{\theta}_{ji}]$ are linear functions, we see that

$$\begin{aligned} a &= f(s;\Theta) + \epsilon(s;\hat{\Theta}) \\ &= \Theta s + \hat{\Theta} s \\ &= (\Theta + \hat{\Theta}) s, \end{aligned} \tag{6.8}$$

which shows that direct parameter perturbation methods (cf. Equation (5.12)) are a special case of SDE and can be expressed in this more general framework.

In effect, state-dependent exploration can be seen as a converter from action-exploring to parameter-exploring methods. A method equipped with the SDE converter does not benefit from all the advantages mentioned in Section 6.2, e.g. actions are not chosen from the same family of functions, since the exploration value is still added to the greedy action. It does, however, cause smooth trajectories and thus mitigates the credit assignment problem (as illustrated in Figure 14).

6.3.3 Adaptive Exploration Variance

For a linear exploration function $\epsilon(s;\hat{\Theta}) = \hat{\Theta}s$ it is also possible to calculate the derivative of the log likelihood with respect to the variance. This allows the system to automatically adapt the amount of exploration, following the same optimisation gradient. Not only will this slowly reduce the variance towards the end of the learning process, when less exploratory behaviour is required, it can also adjust exploration for each parameter dimension individually. This improves the convergence because some parameters may converge quicker than others and therefore require different exploration variances.

First, we need the distribution of the action vector elements a_j:

$$a_j = f_j(s,\Theta) + \hat{\Theta}_j s = f_j(s,\Theta) + \sum_i \hat{\theta}_{ji} s_i \tag{6.9}$$

with $f_j(s,\Theta)$ as the jth element of the return vector of the deterministic controller f and $\hat{\theta}_{ji} \sim \mathcal{N}(0,\hat{\sigma}_{ji}^2)$. We now use two well-known properties of normal distributions: First, if X and Y are two independent random variables with $X \sim \mathcal{N}(\mu_a,\sigma_a^2)$ and $Y \sim \mathcal{N}(\mu_b,\sigma_b^2)$ then $U = X+Y$ is normally distributed with $U \sim \mathcal{N}(\mu_a+\mu_b,\sigma_a^2+\sigma_b^2)$. Second, if $X \sim \mathcal{N}(\mu,\sigma^2)$ and $a,b \in \mathbb{R}$, then $aX+b \sim \mathcal{N}(a\mu+b,(a\sigma)^2)$.

Applied to (6.9), we see that $\hat{\theta}_{ji}s_i \sim \mathcal{N}(0,(s_i\hat{\sigma}_{ji})^2)$, that the sum is distributed as $\sum_i \hat{\theta}_{ji}s_i \sim \mathcal{N}(0,\sum_i(s_i\hat{\sigma}_{ji})^2)$, and that the action element a_j is therefore distributed as

$$a_j \sim \mathcal{N}\Big(f_j(s,\Theta), \sum_i (s_i\hat{\sigma}_{ji})^2\Big), \tag{6.10}$$

where we will substitute $\mu_j := f_j(s,\Theta)$ and $\sigma_j^2 := \sum_i(s_i\hat{\sigma}_{ji})^2$.

Therefore, differentiation of the policy with respect to the free parameters $\hat{\sigma}_{ji}$ yields:

$$\begin{aligned} \frac{\partial \log \pi(a|s)}{\partial \hat{\sigma}_{ji}} &= \sum_k \frac{\partial \log \pi_k(a_k|s)}{\partial \sigma_j} \frac{\partial \sigma_j}{\partial \hat{\sigma}_{ji}} \\ &= \frac{(a_j-\mu_j)^2 - \sigma_j^2}{\sigma_j^4}\, s_i^2 \hat{\sigma}_{ji}, \end{aligned} \tag{6.11}$$

which can directly be inserted into the gradient estimator of REINFORCE. For more complex exploration functions, calculating the exact derivative for the sigma adaptation might not be possible and heuristic or manual adaptation (e.g. with slowly decreasing $\hat{\sigma}$) is required.

6.3.4 Stochastic Policies

The original policy gradient setup as presented in e.g. (Williams, 1992) conveniently unifies the two stochastic features of the algorithm: the stochastic exploration and the stochasticity of the policy itself. Both were represented by the Gaussian noise added on top of the controller. While elegant on the one hand, it also conceals the fact that there are two different stochastic processes. With SDE, randomness has been taken out of the controller completely and is represented by the separate exploration function. So if learning is switched off, the controller only returns deterministic actions. But in many scenarios the best policy is necessarily of stochastic nature.

It is possible and straight-forward to implement SDE with stochastic policies, by combining both random and state-dependent exploration in one controller, as in

$$\boldsymbol{a} = f(\boldsymbol{s}; \boldsymbol{\theta}) + \boldsymbol{\epsilon} + \hat{\boldsymbol{\epsilon}}(\boldsymbol{s}; \hat{\boldsymbol{\theta}}), \tag{6.12}$$

where $\epsilon_j \sim N(0, \sigma_j)$ and $\hat{\theta}_j \sim N(0, \hat{\sigma}_j)$. Since the respective noises are simply added together, none of them affects the derivative of the log-likelihood of the other and σ and $\hat{\sigma}$ can be updated independently. In this case, the trajectories through state-action space would look like a noisy version of Figure 14, bottom plot.

6.3.5 Negative Variances

For practical applications, we also have to deal with the issue of negative variances. Obviously, we must prevent σ from falling below zero, which can happen since the right side of (6.5) can become negative. We therefore designed the following smooth, continuous function and its first derivative:

$$\operatorname{expln}(\sigma) = \begin{cases} \exp(\sigma) & \sigma \leq 0 \\ \ln(\sigma + 1) + 1 & \text{else} \end{cases} \tag{6.13}$$

$$\operatorname{expln}'(\sigma) = \begin{cases} \exp(\sigma) & \sigma \leq 0 \\ \frac{1}{\sigma + 1} & \text{else} \end{cases} \tag{6.14}$$

Substitution of $\sigma^* := \operatorname{expln}(\sigma)$ will keep the variance above zero (exponential part) and also prevent it from growing too fast (logarithmic part). In order to use this substitution, the derivatives in (6.5) and (6.11) have to be multiplied by $\operatorname{expln}'(\sigma)$. In the experiments in section 6.4, this factor is included.

6.3.6 State-Dependent Exploration for Value-Based RL

State-Dependent Exploration has been formulated here for direct Reinforcement Learning, in particular as an extension to William's REINFORCE algorithm. The basic ideas, however, can be carried over to value-based Reinforcement Learning (Section 5.3) as well. Two aspects need to be handled differently: Value-based RL has a value estimator instead of a parameterised controller, and the actions are usually discrete, with the exception of the methods presented in Section 5.3.2. The latter algorithms can be handled the same way as the continuous-action direct RL methods above, using a separate exploration generator that takes the state as input, as shown in Figure 15, bottom graphic.

As a quick reminder for how discrete value-based RL determines the action to take in a given state: The value estimator usually consists of a simple table that stores the value for each state/action combination. To find the greedy action that needs to be executed, a row look-up[2] for the given state, followed by a maximum operation over that row, is performed. The winning cell represents the action to execute.

State-Dependent exploration requires that exploration happens at the beginning of an episode, then is held constant throughout the episode. Furthermore, running into the same state during one episode needs to cause the exact same action to be taken. It would be tedious to choose a random action for each new state, then remember the action throughout the episode for each state. There is a much simpler way to achieve this: At the beginning of an episode, we store the original value table, then permute the columns for each row of the table. Because the maximum is now at a different index for each such permuted row, a seemingly random action is executed instead of the original action. The chosen action is consistent with each state because the table is fixed during the episode. After the episode is completed, the original table can be restored.

Continuous SDE used the variance of the Gaussian distribution to control the amount of exploration. For more fine-grained control for discrete SDE (rather than just *on* or *off*), we can employ an ϵ-greedy scheme: Choose $0 \leq \epsilon \leq 1$, then iterate through each row of the value table and draw a random number $r \sim U(0,1)$ between 0 and 1. If $r < \epsilon$, permute the columns of that row. If not, leave it as is. Over time, ϵ can be reduced from 1 (maximum exploration) towards 0 (no exploration).

State-Dependent Exploration for value-based algorithms will play an important role in the final Chapter of Part II: Context Learning. We finish

2 We assume that states are arranged in rows and actions are arranged in columns.

this chapter with the results of experiments comparing SDE's performance to random exploration.

6.4 EXPERIMENTS

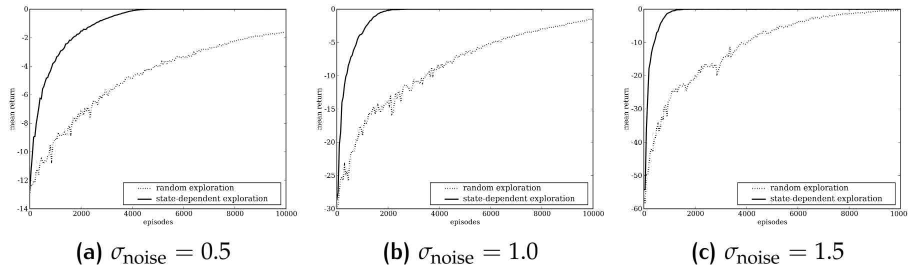
Two different sets of experiments are conducted to investigate both the theoretical properties and the practical application of SDE. The first looks at plain function minimisation and analyses the properties of SDE compared to random exploration (REX). The second demonstrates SDE's usefulness for real-world problems with a simulated robot hand trying to catch a ball.

As SDE was specifically designed for problem domains which do not lend themselves for parameter exploration (see Section 6.2), for example local optimization, or tasks where system knowledge is insufficient or unavailable, direct comparison between parameter-exploring alternatives, like PGPE or neuro-evolution are not in scope of this work.

6.4.1 Function Minimisation

The following sections compare SDE and random exploration (REX) with regard to sensitivity to noise, episode length, and parameter dimensionality. We chose a very basic setup where the task was to minimise $g(x) = x^2$. This is sufficient for first convergence evaluations since policy gradients are known to only converge locally. The agent's state x lies on the abscissa, its action is multiplied with a step-size factor s and the result is interpreted as a step along the abscissa in either direction. To make the task more challenging, we always added random noise to the agent's action. Each experiment was repeated 30 times, averaging the results. Our experiments were all episodic, with the return R for each episode being the average reward as stated in section 5.1. The reward per time step was defined as $r_t = -g(x_t)$, thus a controller reducing the costs (negative reward) minimises $g(x)$.

For a clean comparison of SDE and REX, we used the SDE algorithm in both cases, and emulated REX by drawing new exploration function parameters $\hat{\boldsymbol{\theta}}$ after each step (see Section 6). Unless stated otherwise, all experiments were conducted with a REINFORCE gradient estimator with optimal baseline (Peters and Schaal, 2006) and the following parameters: learning rate $\alpha = 0.1$, step-size factor $s = 0.1$, initial parameter $\theta_0 = -2.0$, episode length $\mathrm{EL} = 15$ and starting exploration noise $\hat{\sigma} = e^{-2} \approx 0.135$.



(a) $\sigma_{\text{noise}} = 0.5$ (b) $\sigma_{\text{noise}} = 1.0$ (c) $\sigma_{\text{noise}} = 1.5$

Figure 16: Convergence for different levels of noise, averaged over 30 runs per curve. The upper solid curve shows SDE, the lower dotted curve REX.

Noise Level

First, we investigated how both SDE and REX deal with noise in the setting. We added normally distributed noise with variance σ^2_{noise} to each new state after the agent's action was executed: $x_{t+1} = x_t + su_t + \mathcal{N}(0, \sigma^2_{\text{noise}})$, where s is the step-size factor and u_t is the action at time t. The results of experiments with three different noise levels are given in Figure 16 and the right part of Table 1.

The results show that SDE is much more robust to noise, since its advantage over REX grows with the noise level. This is a direct effect of the credit assignment problem, which is more severe as the randomness of actions increases.

An interesting side-effect can also be found when comparing the convergence times for different noise levels. Both methods, REX and SDE, ran at better convergence rates with higher noise. The reason for this behaviour can be shown best for a one-dimensional linear controller. In the absence of (environmental) noise, we then have:

$$
\begin{aligned}
x_t &= x_{t-1} + su_{t-1} \\
u_t &= \theta x_t + \epsilon_{\text{explore}}
\end{aligned}
$$

Adding noise to the state update results in

$$
\begin{aligned}
x'_t &= x_t + \epsilon_{\text{noise}} = x_{t-1} + su_{t-1} + \epsilon_{\text{noise}} \\
u'_t &= \theta(x_{t-1} + su_{t-1} + \epsilon_{\text{noise}}) + \epsilon_{\text{explore}} \\
&= \theta(x_{t-1} + su_{t-1}) + \theta\epsilon_{\text{noise}} + \epsilon_{\text{explore}} \\
&= \theta x_t + \epsilon'_{\text{explore}}
\end{aligned}
$$

with $\epsilon'_{\text{explore}} = \theta\epsilon_{\text{noise}} + \epsilon_{\text{explore}}$. In our example, increasing the environmental noise was equivalent to increasing the exploratory noise of the agent, which obviously accelerated convergence.

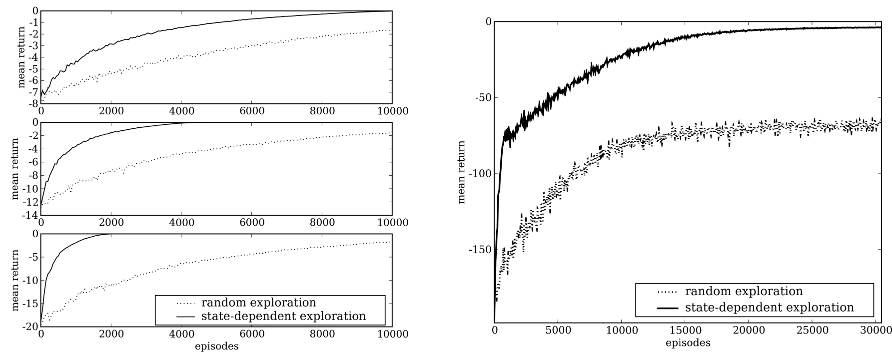


Figure 17: Left: Results for different episode lengths, from top to bottom: 5, 15, 30. Right: Nonlinear controller with 18 free parameters on a 2-dimensional task. With REX, the agent became stuck in a local optimum, while SDE found the same optimum about 15 times faster and then converged to a better solution.

Episode Length

In this series of experiments, we varied only the episode length and otherwise used the default settings with $\sigma_{\text{noise}} = 0.5$ for all runs. The results are shown in Figure 17 on the left side and Table 1, left part. Convergence speed with REX only improved marginally with longer episodes. The increased variance introduced by longer episodes almost completely outweighed the higher number of samples for a better gradient estimate. Since SDE does not introduce more noise with longer episodes during a single roll-out, it could profit from longer episodes enormously. The speed-up factor rose almost proportionally with the episode length.

Table 1: Noise and episode length (EL) sensitivity of REX and SDE. σ_{noise} is the standard deviation of the environmental noise. The steps designate the number of episodes until convergence, which was defined as $R_t > R_{\text{lim}}$ (a value that all controllers reached). The quotient REX/SDE is given as a speed-up factor.

			# steps		
σ_{noise}	EL	R_{lim}	REX	SDE	speed-up
	5		9450	3350	2.82
	15		9150	1850	4.95
0.5	30	-1.8	9700	1050	9.24
	45		8800	650	13.54
	60		8050	500	16.10
0.5			9950	2000	4.98
1.0	15	-1.6	9850	1400	7.04
1.5			7700	900	8.56 899

Parameter Dimensionality

Here, we increased the dimensionality of the problem in two ways: Instead of minimising a scalar function, we minimised $g(x, y) = [x^2, y^2]^T$. Further, we used a nonlinear function approximator, namely a multilayer perceptron with 3 hidden units with sigmoid activation and a bias unit connected to hidden and output layer. We chose a single parameter $\hat{\sigma}$ for exploration variance adaptation. Including $\hat{\sigma}$ the system consisted of 18 adjustable parameters, which made it a highly challenging task for policy gradient methods. The exploration variance was initially set to $\hat{\sigma} = -2$ which corresponds to an effective variance of ~ 0.135. The parameters were initialised with $\theta_i \in [-1, 0[$ because positive actions quickly lead to high negative rewards and destabilised learning. For smooth convergence, the learning rate $\alpha = 0.01$ needed to be smaller than in the one-dimensional task.

As the right-hand side of Figure 17 shows, the agent with REX became stuck after $15,000$ episodes at a local optimum around $R = -70$ from which it could not recover. SDE on the other hand found the same local optimum after a mere $1,000$ episodes, and subsequently was able to converge to a much better solution.

6.4.2 Catching a Ball

This series of experiments is based on a simulated robot hand with realistically modelled physics. We chose this experiment to show the predominance of SDE over random exploration, especially in a realistic robot task. We used the Open Dynamics Engine[3] to model the hand, arm, body, and object. The arm has 3 degrees of freedom: shoulder, elbow, and wrist, where each joint is assumed to be a 1D hinge joint, which limits the arm movements to forward-backward and up-down. The hand itself consists of 4 fingers with 2 joints each, but for simplicity we only use a single actor to move all finger joints together, which gives the system the possibility to open and close the hand, but it cannot control individual fingers. These limitations to hand and arm movement reduce the overall complexity of the task while giving the system enough freedom to catch the ball. A 3D visualisation of the robot attempting a catch is shown in Fig. 18. First, we used REINFORCE gradient estimation with optimal baseline and a learning rate of $\alpha = 0.0001$. We then repeated the experiment with Episodic Natural Actor-Critic (ENAC), to see if SDE can be used for different gradient estimation techniques (Amari, 1998) as well.

3 The Open Dynamics Engine (ODE) is an open source physics engine, see `http://www.ode.org/` for more details.

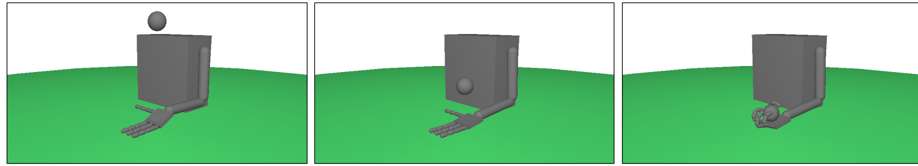


Figure 18: Visualisation of the simulated robot hand while catching a ball. The ball is released 5 units above the palm, where the palm dimensions are 1 x 0.1 x 1 units. When the fingers grasp the ball and do not release it throughout the episode, the best possible return (close to -1.0) is achieved.

Experiment setup

The information given to the system are the three coordinates of the ball position, so the robot "sees" where the ball is. It has four degrees of freedom to act, and in each time step it can add a positive or negative torque to the joints. The controller therefore has 3 inputs and 4 outputs. We map inputs directly to outputs, but squash the outgoing signal with a tanh-function to ensure output between -1 and 1.

The reward function is defined as follows: upon release of the ball, in each time step the reward can either be -3 if the ball hits the ground (in which case the episode is considered a failure, because the system cannot recover from it) or else the negative distance between ball centre and palm centre, which can be any value between -3 (we capped the distance at 3 units) and -0.5 (the closest possible distance considering the palm heights and ball radius). The return for a whole episode is the mean over the episode: $R = \frac{1}{N}\sum_{n=1}^{N} r_t$. In practice, we found an overall episodic return of -1 or better to represent nearly optimal catching behaviour, considering the time from ball release to impact on palm, which is penalised with the capped distance to the palm centre.

One attempt at catching the ball was considered to be one episode, which lasted for 500 time steps. One simulation step corresponded to 0.01 seconds, giving the system a simulated time of 5 seconds to catch and hold the ball.

For the policy updates, we first executed 20 episodes with exploration and stored the complete history of states, actions, and rewards in an episode queue. After executing one learning step with the stored episodes, the first episode was discarded and one new roll-out was executed and added to the front of the queue, followed by another learning step. With this "online" procedure, a policy update could be executed after each single step, resulting in smoother policy changes. However, we did not evaluate each single policy but ran every twentieth a few times

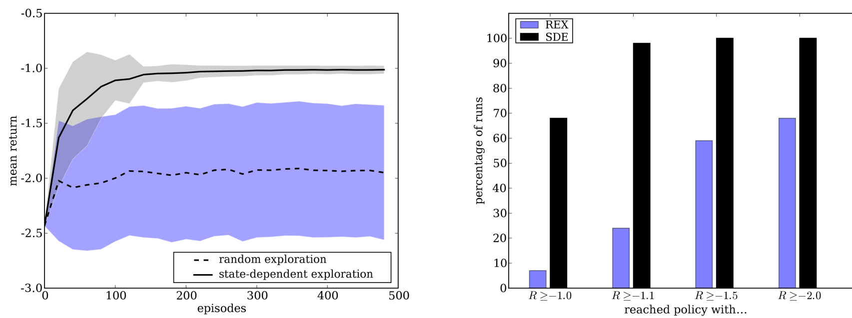


Figure 19: Results after 100 runs with REINFORCE. Left: The solid and dashed curves show the mean over all runs, the filled envelopes represent the standard deviation. While SDE (solid line) managed to learn to catch the ball quickly in every single case, REX occasionally found a good solution but in most cases did not learn to catch the ball. Right: Cumulative number of runs (out of 100) that achieved a certain level. $R \geq -1$ means "good catch", $R \geq -1.1$ corresponds to all "catches" (closing the hand and holding the ball). $R \geq -1.5$ describes all policies managing to keep the ball on the hand throughout the episode. $R \geq -2$ results from policies that at least slowed down ball contact to the ground. The remaining policies dropped the ball right away.

without exploration. This yields a return estimate for the deterministic policy. Training was stopped after 500 policy updates.

6.5 DISCUSSION OF RESULTS

We will first describe the results with REINFORCE. The whole experiment was repeated 100 times. The left side of Figure 19 shows the learning curves over 500 episodes. Please note that the curves are not perfectly smooth because we only evaluated every twentieth policy. As can be seen, SDE finds a near-perfect solution in almost every case, resulting in a very low variance. The mean of the REX experiments indicate a semi-optimal solution, but in fact some of the runs found a good solution while others failed, which explains the high variance throughout the learning process.

The best controller found by SDE yielded a return of -0.95, REX reached -0.97. While these values do not differ much, the chances of producing a good controller are much higher with SDE. The right plot in Figure 19 shows the percentage of runs where a solution was found that was better than a certain value. Out of 100 runs, REX only found a mere 7 policies that qualified as "good catches", where SDE found 68. Almost all SDE runs, 98%, produced rewards $R \geq -1.1$, corresponding to behaviour

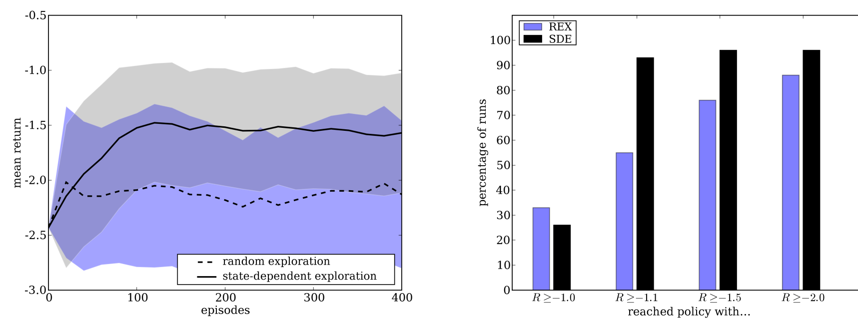


Figure 20: Results after 100 runs with ENAC. Both learning curves had relatively high variances. While REX often did not find a good solution, SDE found a catching behaviour in almost every case, but many times lost it again due to continued exploration. REX also found slightly more "good catches" but fell far behind SDE considering both "good" and "average" catches.

that would be considered a "catch" (closing the hand and holding the ball), although not all policies were as precise and quick as the "good catches". A typical behaviour that returns $R \simeq -1.5$ can be described as one that keeps the ball on the fingers throughout the episode but has not learned to close the hand. $R \simeq -2.0$ corresponds to a behaviour where the hand is held open and the ball falls onto the palm, rolls over the fingers and is then dropped to the ground. Some of the REX trials were not even able to reach the -2.0 mark. A typical worst-case behaviour is pulling back the hand and letting the ball drop to the ground immediately.

To investigate if SDE can be used with different gradient estimation techniques, we ran the same experiments with ENAC (Peters et al., 2005) instead of REINFORCE. We used a learning rate of 0.01 here, which lead to similar convergence speed. The results are presented in Figure 20. The difference compared to the results with REINFORCE is, that both algorithms, REX and SDE had a relatively high variance. While REX still had problems to converge to stable catches (yet showed a 26% improvement over the REINFORCE version of REX for "good catches"), SDE in most cases (93%) found a "catching" solution but often lost the policy again due to continued exploration, which explains its high variance. Perhaps this could have been prevented by using tricks like reducing the learning rate over time or including a momentum term in the gradient descent. These advancements, however, are beyond the scope of this paper. SDE also had trouble reaching near-optimal solutions with $R \geq -1.0$ and even fell a little behind REX. But when considering policies with $R \geq -1.1$, SDE outperformed REX by over 38%. Overall the

experiments show that SDE can in fact improve more advanced gradient estimation techniques like ENAC.

7 SEQUENTIAL FEATURE SELECTION

7.1 INTRODUCTION

In recent times, an enormous increase in data has been observed, without a corresponding growth of the information contained within them. In other words, the *redundancy* of data continuously increases. An example of such effects can be found in medical imaging. Diagnostic methods can be improved by increasing the amount of MRI, CT, EMG, and other imaging data yet the amount of underlying information does not increase. Even worse, the redundancy of such data seems to negatively impact the performance of associated classification methods. Indeed, common engineering practices employ data-driven methods (including dimensionality reduction, nonlinear PCA, etc.) to reduce data redundancy.

On the other hand, obtaining qualitatively good data gets increasingly expensive. Again, medical data serves as a good example: not only do the costs of the above-mentioned medical imaging techniques explode—MRT scans are performed at the end user price of several thousands of US dollars per hour—but also diagnostics tests are getting increasingly intricate and therefore costly, to the point that a selection of the right diagnostic methods while maintaining the level of diagnostic certainty is of high value.

Also, from a computer scientist's perspective, the amount of processable data grows faster than processor speed. According to various studies[1], recent years showed an annual 40–60% increase of commercial storage needs and a 40+-fold increase is expected in the next decade. Though this may, just like the integration density of processors, follow Moore's law, the increase of computer speed is well below that.

In short, an improved approach *feature selection* (FS) is needed, which not only optimally spans the input space, but optimises with respect to data consumption. All of these arguments clearly demonstrate the advantage of carefully selecting relevant portions of data. Going beyond traditional FS methods, in this paper we lay out and demonstrate an approach of selecting features in sequence, making the decision which feature to select next *dependent* on previously selected features and the current internal

1 E.g., Gartner's survey at http://www.gartner.com/it/page.jsp?id=1460213.

state of the supervised method that it interacts with. In particular, our Sequential Online[2] Feature Selection (SOFS) will embed Reinforcement Learning (RL) into classification tasks, with the objective to reduce data consumption and associated costs of features during classification. The general question we would like to answer with this work is: "*Where do I have to look next, in order to keep data consumption and expenses low while maintaining high classification results?*"

In this chapter, we will derive common supervised learning tasks, in this case multi-class classification problems, in a sequential manner and thus make them accessible to Reinforcement Learning algorithms. Reinforcement Learning is commonly used for control tasks in robotics (Abbeel et al., 2007), scheduling problems (Zhang and Dietterich, 1995) or game play (Erev and Roth, 1998). By applying it to general supervised learning tasks, however, we aim for a hybrid system that can simultaneously learn an attentive control mechanism to steer focus towards informative portions of data while learning the supervised mapping function. Instead of a static feature selection as a pre-processing step, this approach can be seen as sequential online feature selection, that chooses features on the fly, depending on the current state of the supervised task it interacts with.

this not only leads to greatly reduced data consumption in various experiments, speeding up the learning process and improving the results, it also proves useful in real-world scenarios where data is not a free resource but its acquisition is time-consuming and/or costly.

Background literature and previous related work is discussed in 7.2. The SOFS framework is mapped out in Section 7.3. We then formally define sequential classifiers and rephrase the problem as a Partially Observable Markov Decision Processes (POMDP). In addition, a novel action selection mechanism without replacement is introduced. Section 7.5 then demonstrates our approach, first on two artificially created toy examples, then on real-world problems, both with redundant (handwritten digit classification) and costly (diabetes classification) data and discusses the results.

7.2 BACKGROUND

Feature selection (Liu and Motoda, 2008) with RL has been addressed previously (Gaudel and Sebag, 2010), yet the novelty of our approach

2 We will occasionally omit the word "Online" for brevity, but all references to this algorithm always select features online, on the fly, while sequentially classifying the input sample.

lies in its sequential decision process. Our work is based on and inspired by existing research, combining aspects of online FS (Wu et al., 2010; Perkins and Theiler, 2003) and attentional control policy learning (Schmidhuber and Huber, 1991; Paletta and Pinz, 2000; Paletta et al., 2005; Bazzani et al., 2011). A similar concept, Online Streaming FS (Wu et al., 2010) has features streaming in one at a time, where the control mechanism can accept or reject the feature. While we adopt the idea of sequential feature selection, our scenario differs in that it allows access to all features with the sub-goal of minimising data consumption.

A closely related work (Dulac-Arnold et al., 2011b,a) classifies inputs in sequential manner based on approximate policy iteration (API). Their work describes a more integrated approach where both classification and feature selection is done by a single component, using the features directly rather than the classifier belief. They solve the classification problem by giving the agent not only action choices about selecting new features, but also classifying the the sample into any of the available classes. The mapping of state-action pairs to values is implemented by a linear regression that uses block vector coding to encode actions into the state-action space.

Feature selection on medical data specifically has been published before (Raymer et al., 2003), using a hybrid Naive Bayes classifier/Evolutionary Algorithm. However, this line of work follows the more traditional approach of extracting features as a separate preliminary step before classification, therefore yielding significantly lower overall results than the approach described here.

Related work further includes a system that uses RL for feature learning in object tracking dependent on visual context (Liu and Su, 2004; Harandi et al., 2004), and an approach where RL is used to create an ordered list of image segments based on their importance for a face recognition task (Norouzi et al., 2010). However, their decision process is not dependent on the internal state of the classifier, which brings their method closer to conventional FS. Other approaches describe Reinforcement Learning on raw pixel data (Ernst et al., 2006), circumventing feature selection all together.

Section 7.5.3 includes experiments manipulating an attentive vision system across images of hand-written digits. While quite different in their approach, Srivastava et al. (2012, 2013) describe some interesting commonalities in their work. They, too, control a vision system (in their case, a fovea with decreasing resolution towards the edges) over images, manipulated by a learned controller. However, their motivation is a different one. Instead of minimizing data consumption, their algorithm PowerPlay learns to self-invent new problems and modifications to the existing controller that can solve these new problems, thus becoming an

increasingly general problem solver. Another similarity to this work is the ability of their SLIM NN controller networks (Schmidhuber, 2012) to learn to stop the computation of a program at any time. We discuss this problem and our solution to self-halting computation in Section 7.4. Quantitative evaluations between the two methods were not conducted due to the very different optimization objective (data minimization vs. skill generalization).

7.3 SEQUENTIAL ONLINE FEATURE SELECTION

7.3.1 General Idea

In Machine learning, solving a classification problem means to map an input x to one of a finite set of class labels $\mathcal{C}$. Classification algorithms are trained on labelled training samples $I = \{(x^1, c^1), \ldots, (x^n, c^n)\}$, while the quality of such a learned algorithm is determined by the generalisation error on a separate test set. We regard features as disjunct portions (scalars or vectors) of the input pattern x, with feature labels $f_i \in F$ and feature values $f_i(x)$ for feature f_i. One key ingredient for good classification results is feature selection (also called *feature subset selection*): filtering out irrelevant, noisy, miss-leading or redundant features. FS is therefore a combinatorical optimisation problem that tries to identify those features which will minimise the generalisation error. In particular, FS tries to reduce the amount of useless or redundant data to process.

We wanted to take this concept even further and focus on minimising *data consumption*, as outlined in the introduction. For this purpose, however, FS is not ideal. Firstly, the FS process on its own commonly assumes free access to the full dataset, which defeats the purpose of minimising data access in most real-world scenarios. But more significantly, FS determines for *any* input the *same subset* of features that should be used for a subsequent classification. We argue that this limitation is not only unnecessary, but in fact disadvantageous in terms of minimising data consumption.

We believe that by turning classification into a sequential decision process, we can further reduce the amount of data to process significantly, as FS and classification then become a closely intertwined process: deciding which feature to select next depends on the previously-selected features and the behaviour of the classifier on them. This will be achieved by using a fully trained classifier as an environment for a RL agent, that

learns which feature to access next, receiving reward on successful classification of the partially uncovered input pattern.

7.3.2 Application Scenarios

We will cover three application scenarios, which scientists and engineers interested in this research may most likely face:

Scenario 1: access to pre-trained classifier

In this scenario, a pre-trained classifier K exists, presumably designed and tuned carefully with much effort. While we can query K for classifications, re-training K, or training a new classifier, is not an option. The goal in this scenario is to use the pre-trained classifier K in the sequential decision process to train a SOFS agent that utilises the existing K while learning to access as few features as possible on future classifications (test dataset). In this scenario, K acts as a fixed black-box environment for the agent and will not be modified.

Scenario 2: access to training data

This scenario assumes, that a complete training dataset is available, i.e. for every sample *all* features are available. The used resources (time and costs) to acquire this dataset are irrelevant (i.e. already spent), and may as well be used to train the agent. The task is two-fold: First, train a classifier $\widetilde{K}$ on the full dataset. Unlike Scenario 1, we can train $\widetilde{K}$ already in sequential manner, which will lead to better performance during RL training. The second task is to train the SOFS agent to minimise data consumption and expenditure on future classification (test dataset) with $\widetilde{K}$. A typical example of this scenario is in the field of medical trials, where previous experiments with patients have been collected. The time and money to conduct these experiments has been spent already but may help to reduce the required features and therefore the costs for future experiments.

Scenario 3: limited access to data source

In the third scenario, we have access to a data source, but requesting samples (or requesting the correct label for a sample) is time-consuming and/or costly, even for training the classifier. The goal is to use as few data samples as possible to both train a sequentialized classifier $\widetilde{K}$ and the SOFS agent simultaneously. This scenario represents cases where

any access to data is associated with some costs and no previously trained classifier (Scenario 1) or training dataset (Scenario 2) is available. As an example, think of a satellite that has been launched to earth's orbit to take pictures of distant galaxies. To classify the star formations, data has to be sent through an expensive relay station, where each megabyte costs hundreds of dollars. The goal is to minimise data consumption already during training for both the classifier and the SOFS agent.

While these three scenarios cover many use cases and deserve special attention, the presented approach is of general nature and broadly applicable beyond the here mentioned use cases.

7.3.3 Additional Notation

For the next sections, we additionally require the following notation: ordered sequences are denoted $(\cdot)$, unordered sets are denoted $\{\cdot\}$, appending an element e to a sequence s is written as $s \circ e$. Related to power sets, we define a *power sequence* $\text{powerseq}(M)$ of a set M to be the set of all ordered permutations of all elements of the power set of M, including the empty sequence $()$. As an example, for $M = \{1, 2\}$, the resulting $\text{powerseq}(M) = \{(), (1), (2), (1, 2), (2, 1)\}$. During an episode, the feature history $h_t \in \text{powerseq}(F)$ is the sequence of all previously selected features in an episode up to and including the current feature at time t. To unclutter the notation, we will introduce the symbol v_t, which represents the sequence of values of a feature history h_t, as follows:

$$v_t = (h_\tau(x))_{0<\tau\leq t} = (h_0(x), h_1(x), \ldots, h_t(x)) \tag{7.1}$$

Costs associated with accessing a feature f are represented as negative scalars $r_f^- \in \mathbb{R}, r_f^- < 0$. We further introduce a non-negative global reward $r^+ \in \mathbb{R}, r^+ \geq 0$ for correctly classifying an input. A classifier in general is denoted with K, and sequential classifiers (defined in Section 7.3.4) are written as $\widetilde{K}$.

7.3.4 Sequential Classification

We define a *sequential* classifier $\widetilde{K}$ to be a functional mapping from the power sequence of feature values to a set of classes:

$$\widetilde{K} : \text{powerseq}\left(\{f(x)\}_{f \in F}\right) \rightarrow \mathcal{C} \tag{7.2}$$

Our framework assumes that feature values are passed to the classifier $\widetilde{K}$ one at a time, therefore $\widetilde{K}$ requires some sort of memory. Recurrent

Neural Networks (RNN) (Hüsken and Stagge, 2003), for instance, are known to have implicit memory that can store information about inputs seen in the past. If the classifier does not possess such a memory, it can be provided explicitly: at time step t, instead of presenting only the t-th feature value $f_t(x)$ to the classifier, the whole history $(f_1(x), \ldots, f_t(x))$ up to time t is presented instead.

As it turns out, the above approach of providing explicit memory can also be used to turn any classifier, that can handle *missing values* (Saar-Tsechansky and Provost, 2007), into a sequential classifier. For a given input x and a set F_1 of selected features, $F_1 \subseteq F$, the values of the features not chosen, i.e. $F \backslash F_1$, are defined as *missing*, which we will denote as ϕ. Each episode starts with a vector of only missing values $(\phi, \phi, \ldots)$, where ϕ can be the mean over all values in the dataset, or simply consist of all zeros. More sophisticated ways of dealing with missing values based on imputation methods (Saar-Tsechansky and Provost, 2007) can be implemented accordingly. At each time step, the current feature gradually uncovers the original pattern x more. As an example, assuming scalar features f_1, f_4 and f_6 were selected from an input pattern $x \in \mathbb{R}^6$, the input to the classifier K would then be: $(f_1(x), \phi, \phi, f_4(x), \phi, f_6(x))$. This method allows us to use existing, pre-trained non-sequential classifiers in a sequential manner. Note, that the classifiers will remain unchanged and only act as an environment in which the SOFS agent learns. We therefore do not measure the performance of the classifiers but rather the number of features necessary until correct classification was achieved.

7.3.5 Classifier State Representation

In order to train a Reinforcement Learning agent based on the state of the classifier, we need to find a suitable representation of such a state. This representation is rather arbitrary, but it needs to summarise the current "situation", in which the classifier can find itself. Ideally, the representation should be consistent, so that for similar (or even identical) situations, the state should be similar (or identical) as well. For Reinforcement Learning algorithms to work best, the state should also be as low-dimensional as possible without affecting performance.

Instead of simply stating the representation that was eventually used for this task, we will quickly describe the thought process that went into deciding on a suitable representation.

Let's start off with a suggestion for a classifier state representation: take all the available information of the classifier (inputs, outputs, architecture, parameters, etc), convert them to binary code and use that (pre-

sumably very long) bit string as state for the agent. This is a very bad example for a classifier representation, because it is very high-dimensional, and a small change in the bit representation could mean a big change in the classifier (e.g., if some of its parameters are affected). And while one could certainly implement a system with this representation, the RL agent would most likely not be able to learn anything with it.

What would be a more reasonable state representation? Maybe we should encode what the classifier sees, which is the features that have been uncovered so far and their values. Thus, the classifier state could be encoded as a vector of length $2n$, where n is the number of available features. The first n elements would encode the values of already accessed features (the values for not yet selected features would be set to the missing value ϕ), whereas the last n elements of the vector would represent the features that have been selected already, encoded as bits. Now the agent could see what information the classifier gets, and could base its decision on which feature to select next on that representation[3].

However, this representation could still get quite long, especially when dealing with high-dimensional input spaces. For the experiment described in section 7.5.3, we have an input space with 784 features if we regard each pixel as a feature, our representation length would be double that. Another problem with this representation is that the RL agent sees everything the classifier sees, and therefore has to learn everything the classifier has to learn, in addition to the state-action value function, which would be somewhat redundant. And finally, it might perhaps not be necessary to act differently if there is the slightest difference in the input pattern.

The key thought that leads to a much shorter state representation is that samples from the same class all have something in common, and it may perhaps be enough for the agent to act based on these commonalities. Again, consider the MNIST experiment (described in section 7.5.3) where we have hand-written digits on a 28×28 pixel grid. Most 1s will have a vertical stroke (sometimes slightly tilted) in the middle of the grid, but so do 7s. What makes 7s different from 1s is the additional horizontal stroke in the top left corner (see Figure 27, images 2 and 5, for an example). So to distinguish between these two classes, the agent should look in the top left corner.

Therefore, a good state representation that helps the agent quickly distinguish similar classes is one that captures the class belief of the classifier: a vector b of length $|\mathcal{C}|$, the number of classes, where each element of b indicates the probability of the current sample to belong to that class. It

3 In fact, we will use this representation in section 7.4, where we look at how to simultaneously select features and classify with the agent.

is independent of the input dimensionality but still reflects an important property of the classifier.

One problem with such a state representation remains though, which is its ambiguity. The classifier belief state could be the same for very different input samples because the input is not part of its representation. This means that the information accessible to the agent is incomplete. This moves this problem class out of what is covered by regular MDPs and into a problem class called POMDP, *partially observable* MDP. The next section explains, how we can formally define classification based on a belief state representation as POMDP.

7.3.6 Classification as POMDP

We will now re-formulate classification as a Partially Observable Markov Decision Process[4] (POMDP) (Monahan, 1982), making the problem sequential and thus accessible to Reinforcement Learning algorithms (Aberdeen, 2003).

To map the original problem of classification under the objective to minimise data consumption to a POMDP, we define each of the elements of the 6-tuple $(S, A, O, \mathcal{P}, \Omega, \mathcal{R})$, which describes a POMDP, as follows: the state $s \in S$ at time step t comprises the current input x, the classifier $\widetilde{K}$, and the previous feature history h_{t-1}, so that $s_t = (x, \widetilde{K}, h_{t-1})$. This triple suffices to fully describe the decision process at any point in time. Actions $a_t \in A$ are chosen from the set of features $F \backslash h_{t-1}$, i.e. previously chosen features are not available. Section 7.3.7 describes, how this can be implemented practically.

The observation is represented by the classifier's internal belief of the class after seeing the values of all features in h_{t-1}, written as $o_t = b(x, \widetilde{K}, h_{t-1}) = b(s_t)$. Most classifiers base their class decision on some internal belief state. A Feed Forward Network (FFN) for example often uses a soft-max output representation, returning a probability p_i in $[0, 1]$ for each of the classes with $\sum_{i=1}^{|\mathcal{C}|} p_i = 1$. And if this is not the case (e.g. for purely discriminative functions like a Support Vector Machine), a straightforward belief representation of the current class is a k-dimensional vector with a 1-of-k coding. In the experiments section, we will demonstrate examples with FFN, RNN and Naive Bayes classifiers. Each of these architectures allows us to use the aforementioned

4 A *partially observable* MDP is a MDP with limited access to its states, i.e. the agent does not receive the full state information but only an incomplete observation based on the current state.

soft-max belief over the classes as belief state for the POMDP. The probabilities p_i for each class serve as an observation to the agent:

$$o_t = b(x, \widetilde{K}, h_{t-1}) = (p_1, p_2, \dots, p_{|\mathcal{C}|}) \tag{7.3}$$

Assuming a fixed x and a deterministic, pre-trained classifier $\widetilde{K}$, the state and observation transition probabilities $\mathcal{P}$ and Ω collapse and can be described by a deterministic transition function T, resulting in the next state and observation:

$$s_{t+1} = T_x(s_t, a_t) = (x, \widetilde{K}, h_{t-1} \circ a_t) \tag{7.4}$$

$$o_{t+1} = b\left(s_{t+1}\right) \tag{7.5}$$

Lastly, the reward function $\mathcal{R}$ returns the reward r_t at time step t for transitioning from state s_t to s_{t+1} with action a_t. Given c as the correct class label, and the history of all feature values v_t (defined in Eqn. 7.1), the reward is given as:

$$r_t = \begin{cases} r^+ + r^-_{a_t} & \text{if } \widetilde{K}(v_t) = c \\ r^-_{a_t} & \text{else} \end{cases} \tag{7.6}$$

A graphical representation of the information flow for SOFS is depicted in Fig. 21.

7.3.7 Action Selection without Replacement

In this specific task we must ensure that an action (a feature) is only chosen at most once per episode, i.e. the set of available actions at each given decision step is dependent on the history h_t of all previously selected actions in an episode. Note that this does not violate the Markov assumption of the underlying MDP, because no information about available actions flows back into the state and therefore the decision does not depend on the feature history.

Value-based RL offers an elegant solution to this problem. By manually changing all action-values $Q(o, a_t)$ to $-\infty$ after choosing action a_t, we can guarantee that all actions not previously chosen in the current episode will have a larger value and be preferred over a_t. A compatible exploration strategy for this action selection without replacement is Boltzmann exploration. Here, the probability of choosing an action is proportional to its value under the given observation:

$$p(a_t|o_t) = \frac{e^{Q(o_t, a_t)/\tau}}{\sum_a e^{Q(o_t, a)/\tau}}, \tag{7.7}$$

where τ is a temperature parameter that is slowly reduced during learning for greedier selection towards the end. Thus, when selecting action

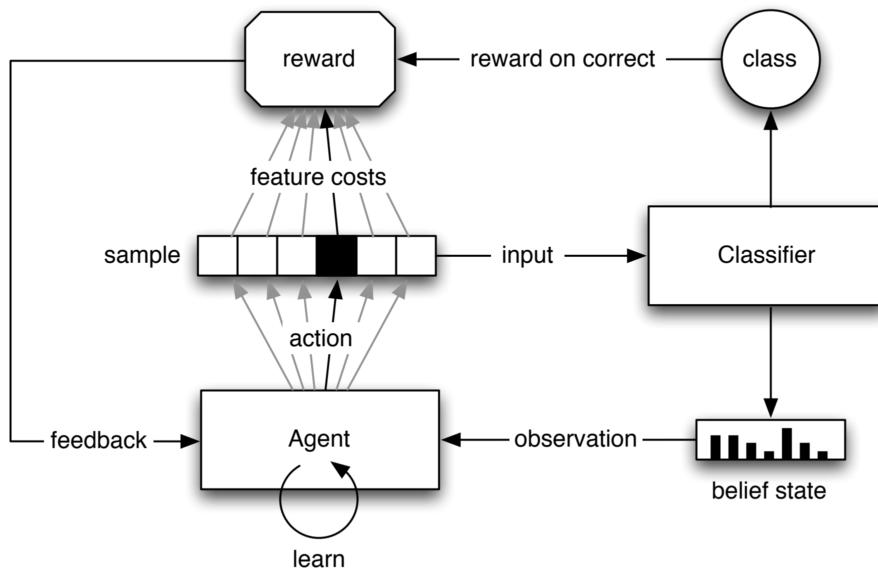


Figure 21: SOFS explained graphically. The data sample (initially consisting of missing values only) is presented to the classifier, which outputs the most likely class and a belief state over all classes. The belief state acts as input to the RL agent, which will choose an action (a feature f) based on the belief. The action results in a feature cost r_f^-. In addition, a positive reward r^+ is added if the classification was correct. The data sample with the now uncovered feature f is fed to the classifier and the process is repeated until correct classification occurred. In the learning phase, the sum of rewards act as feedback to the agent who will then learn to improve its behaviour to gain more reward over time.

a_{t+1}, all actions in h_t have a probability of $e^{-\infty} = 0$ of being chosen again. At the end of an episode, the original Q-values are restored.

7.3.8 Solving the POMDP

Having defined the original task of classification with minimal data consumption as a POMDP and solved the problem of action selection without replacement, we can revert to existing solutions for this class of problems. Since the transition function is unknown to the agent, it needs to learn from experience, and a second complication is the continuous observation space. For regular MDPs, a method well-suited to tackle both of these issues is Fitted Q-Iteration (FQI) (Ernst et al., 2005). The sequential classifier $\widetilde{K}$ then takes care of the PO part of the POMDP, yielding a static belief over the sequential input stream.

FQI uses a batch-trained function approximator (FA) as action-value function. Here, we will use Locally Weighted Projection Regression (LWPR) (Vijayakumar and Schaal, 2000) as the value function approximator of choice, as it is a fast robust online method that can handle large amounts of data.

The details of the algorithm are presented in Listing 2. The history is always initialised with the missing value ϕ (line 3). This gives the system the chance to pick the first feature before seeing any real data. The SOFS agent is trained after every episode (line 16), which ends either with correct classification (line 9–11) or when the whole input pattern was uncovered (line 15), i.e. all features were accessed.

Early on, neither $\widetilde{K}$ nor the agent A will perform well, so the classifier is mostly trained with the complete input x. The better $\widetilde{K}$ and A get, the more frequent the episode is stopped early (line 9–11), and the classifier will be trained with shorter and shorter feature sequences, while the agent learns, in which order the features need to be selected to be classified correctly.

In its present form, the algorithm applies to Scenario 3, where both the classifier and SOFS are trained concurrently. Scenario 1 does not need to train a classifier (instead using the explicit memory procedure from section 7.3.4 on the existing classifier K), therefore line 16 can be skipped. Scenario 2 requires two phases of training. The first phase trains a classifier sequentially on the full dataset I with randomly permuted feature sequences: line 6 is replaced with a random action selection without replacement and line 17 is skipped. After training $\widetilde{K}$, the second phase only trains the agent A (analogous to Scenario 1) by skipping line 16.

7.4 LEARNING WHEN TO STOP

In its current state, SOFS learns an attentive model that picks the next feature, based on expectation to correctly classify the input. There is currently no mechanism that makes the final decision as to which class the sample belongs to. Knowing the ground truth, we can make comparisons how few features have to be compared until the correct class is chosen, but in order to have a full equivalent to a classifying system, one cannot use the ground truth to make this decision. Therefore, a stopping criterion is required, that tells us "I'm confident that I found the correct class". Different strategies were tested in experiments and are discussed below, ranging from naive ideas that failed towards a working solution at the end.

Algorithm 2 Sequential Online Feature Selection (SOFS)

Require: labelled inputs I, agent A, sequential classifier $\widetilde{K}$
1: **repeat**
2: choose $(x, c) \in I$ randomly
3: $h_0 \leftarrow (\phi)$
4: $o_1 \leftarrow b(x, \widetilde{K}, h_0)$
5: **for** $t = 1$ to $|F|$ **do**
6: $a_t \leftarrow \mathrm{A}(o_t)$
7: $h_t \leftarrow h_{t-1} \circ a_t$
8: $o_{t+1} \leftarrow b(x, \widetilde{K}, h_t)$
9: **if** $\widetilde{K}(v_t) = c$ **then**
10: $r_t \leftarrow (r^+ + r^-_{a_t})$
11: **break**
12: **else**
13: $r_t \leftarrow r^-_{a_t}$
14: **end if**
15: **end for**
16: train $\widetilde{K}$ with (v_t, c)
17: train A with $(o_1, a_1, r_1, \ldots, r_t, o_{t+1})$
18: **until** convergence

7.4.1 Potential Stopping Criteria

Fixed Number of Features

A very simple stopping decision only counts the accessed features and stops after a pre-defined number. It would seem that this could be a successful strategy, especially when we are willing to accept a high number of features, possibly close to the total number of features. This is not necessarily the case though, because during learning, the algorithm stops when the class was correctly identified and no subsequent features are uncovered anymore. Imagine a very obvious input sample, that can always be correctly classified after uncovering the first feature. The classifier has never seen the same sample with more than one feature uncovered, and therefore has no experience with the extra information from the other features, which likely leads to a wrong classification.

Belief Threshold

SOFS calculates the classifier belief state over all classes which is a vector of probabilities that the current sample belongs to a given class. This vector is normalised (sums up to 1). The classifier belief could be used

as a stopping condition, if one of the class probabilities exceeds a certain threshold (e.g., 0.7).

Experiments testing this stopping condition failed because the threshold is chosen arbitrarily and there was no incentive for the classifier to maximise the probabilities to be higher than that threshold. For reasonable threshold, the winning class was often above the threshold, but not all the time. Choosing the threshold too low resulted in many false positives, while choosing it too high led to very long episodes that often did not classify at all (because the threshold was never reached).

Supervised Stopping

Another way of deciding when to stop the classification is to use a separate supervised binary classifier that learns whether the process of uncovering features should continue or be stopped and the most probable class be output as winner. The input to such a stopping classifier could be a combination of the following: The already accessed features of the sample, the position of those features, the internal belief state of the classifier, the number of features accessed. While it is not a problem to train such a classifier during SOFS learning, experiments indicated that it is very hard to successfully learn the stopping decision, because in order to know if classification can be stopped, the classifier would have to understand what the SOFS agent has learned. The decision to stop cannot be made independent of the agent's knowledge.

Stop Action

This idea is based on the argument above, that in order to know when to stop classifying, one needs to know the decision process of the SOFS agent. Therefore, we could let the agent make that decision itself.

In addition to the feature actions, the agent has one more action available at each point in time. Choosing this "stop" action indicates that the feature selection process should be stopped and the currently highest probability in the belief state should be returned as the winning class.

This approach assumes, that the RL agent knows, when the uncovered features carry enough information to make the correct decision. But this means that the agent needs to learn the correct classification just as the classifier does, which would make the classifier obsolete. Also, the stop action is fundamentally different from the feature selection actions, which requires a very flexible function approximator for the value

estimation. It seems that the agent just does not have enough predictive power to learn both the feature selection task and the stopping task.

Classifying Actions

It seems that in order to make the stopping decision, all available information about the task has to be held in one place and the decision-making process needs access to everything. This means that the two components—classifier and attentive feature selector—have to be merged into a single learner. This can be done by having a RL setup with two different kinds of actions: Those that choose new features, and those that classify the sample into one of the available classes (and end feature selection).

It turns out that this approach works well and the agent is now able to learn when to uncover more features or when to classify and which class to choose.

7.4.2 Integrated Classification — Reformulation as MDP

In order to create a fully integrated classification system (including a stopping mechanism), we will have to reformulate the whole framework and make some changes. First of all, the agent and the classifier have to be combined into a single decision-making component. Secondly, because the agent is also the classifier, it requires access to the full pattern (or at least the uncovered part of it) instead of just the belief state. This is a grave difference to the earlier implementation because the state space for the RL part is now much larger and the learning problem more complex. On the positive side, this means that the process now becomes Markovian and we can reformulate Sequential Online Feature Selection with Integrated Classification (SOFS+IC) as an MDP:

To map the problem to a MDP, we need to define the 4-tuple $(S, A, \mathcal{P}, \mathcal{R})$, with S being the set of states, A the set of actions, $\mathcal{P}^a_{s,s'}$ the transition probability distribution and $\mathcal{R}^a_{s,s'}$ the reward function when transitioning from s to s' with action a.

We define m_t as the bit mask consisting of 1 for observed features and 0 for unseen features at time step t. The feature pattern vector p_t is the vector of all feature values of uncovered features at time t and 0 everywhere else and can be calculated by element-wise multiplication ($\odot$) of the input pattern x with the bit mask m_t: $p_t = x \odot m_t$ (compare Section

7.3.4). The MDP state $s_t \in S$ at time step t then is the concatenation ($\circ$) of the feature pattern vector p_t and the bit vector m_t,

$$s_t = p_t \circ m_t = (x \odot m_t) \circ m_t, \tag{7.8}$$

with a vector consisting of only zeros at time $t = 0$. In addition, we introduce a terminal state T that indicates that classification took place and the episode is finished. The set S of all possible states is then the set of all states s that can be constructed over the training set I according to Eqn. (7.8) and the terminal state T.

The action set A is the union of all "uncovering" actions (i.e. those that access a new, unseen feature) and all "classification" actions (i.e. those that classify the sample into one of the classes and end the episode).

$$A = \mathcal{C} \cup F \backslash h_{t-1} \tag{7.9}$$

The transition probability $\mathcal{P}^a_{s,s'}$ again collapses to a deterministic transition function and the next state can be expressed as:

$$s_{t+1} = \begin{cases} (x^n \odot m_{t+1}) \circ m_{t+1} & \text{if} \quad a_t \in F \backslash h_{t-1} \\ T & \text{if} \quad a_t \in \mathcal{C} \end{cases} \tag{7.10}$$

Finally, the reward function $\mathcal{R}$ defines new rewards as:

$$r_{t+1} = \begin{cases} r^-_{a_t} & \text{if} \quad a_t \in F \backslash h_{t-1} \\ +r^+ & \text{if} \quad a_t \in \mathcal{C} \text{ and } a_t = c^n \\ -r^+ & \text{if} \quad a_t \in \mathcal{C} \text{ and } a_t \neq c^n \end{cases} \tag{7.11}$$

The algorithm for SOFS+CI is listed as Alg. 3. The feature-uncovering process takes place in line 12 where the a_t-th bit of m_t is switched from 0 to 1 and the feature value for f_{a_t} is then included in the new state in line 13. Note that it is not necessary to know all the values of vector x to calculate s_{t+1} in line 13. The element-wise multiplication with the bit mask is merely a convenient way of expressing the new state mathematically. Also note that the for loop (line 6) is interrupted early (line 22) as soon as a classifying (i.e. non-uncovering) action is chosen, otherwise it is continued in line 14.

With this MDP formulation, we can now solve classification with standard RL algorithms as before, for example Fitted Q-Iteration (FQI). Because the system is now a complete classifier it can be compared to regular classifiers as well. We chose to compare it to a decision tree classifier (specifically C4.5), as these are most similar to the sequential nature of SOFS and we can not only compare its classification accuracy but also how many features each of the methods accessed until correctly classifying the sample. Experiment results based on this implementation on the Cube dataset are presented in Section 7.5.5 and Table 4.

Algorithm 3 SOFS with Integrated Classification (SOFS+IC)

Require: labelled inputs I, agent A

1: **repeat**
2: choose $(x, c) \in I$ randomly
3: $m_1 \leftarrow \vec{0}$
4: $s_1 \leftarrow \vec{0}$
5: $h_0 \leftarrow (\phi)$
6: **for** $t = 1$ to $|F|$ **do**
7: $a_t \leftarrow \mathrm{A}(s_t)$
8: $h_t \leftarrow h_{t-1} \circ a_t$
9: **if** $a_t \in F \backslash h_{t-1}$ **then**
10: $r_t \leftarrow r^-_{a_t}$
11: $m_{t+1} \leftarrow m_t$
12: $m_{t+1,a_t} \leftarrow 1$
13: $s_{t+1} \leftarrow (x \odot m_{t+1}) \circ m_{t+1}$
14: **continue**
15: **end if**
16: **if** $a_t = c$ **then**
17: $r_t \leftarrow r^+$
18: **else**
19: $r_t \leftarrow -r^+$
20: **end if**
21: $s_{t+1} \leftarrow T$
22: **break**
23: **end for**
24: train A with $(s_1, a_1, r_1, \ldots, r_t, s_{t+1})$
25: **until** convergence

7.5 EXPERIMENTS

We evaluate the proposed method on four different datasets to demonstrate and point out certain properties of SOFS: two artificial toy examples, the MNIST handwritten digits classification task, and a medical dataset for diabetes prediction. Each experiment was repeated 25 times, the plots for MNIST and the diabetes task show single runs (grey) and the mean value over all runs (black). Finally, we ran another set of experiments on the Cube dataset with the SOFS+IC algorithm and compare it to a regular decision tree classification with a C4.5 algorithm.

pattern 1			pattern 2			pattern 3			pattern 4		
1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9

Figure 22: Artificial toy data to investigate whether SOFS bases its decision on the current state or simply chooses informative features (like regular FS) independent of the state. Pattern 1 and 2 can be distinguished with features 8 or 9, while pattern 3 and 4 can be distinguished with feature 6.

7.5.1 Toy Example I – Shapes

This toy dataset was inspired by the MNIST handwritten digits set (Section 7.5.3) but is much simpler, has a lower dimension and only 4 different patterns, illustrated in Figure 22. It was chosen to get an insight into the decision making process of a trained SOFS agent, which a large dataset like MNIST cannot provide that easily.

Each pattern consists of 3×3 pixels, and each pixel was considered a feature. We used artificially created training data (1000 samples, each randomly chosen from the four patterns).

We looked at Scenario 2 in this experiment, generating a training dataset of 1000 samples, randomly chosen from the 4 patterns. In the first phase, we trained a FFN classifier with a 9-20-4 architecture, sigmoid activation function in the hidden layer, and soft-max activation in the output layer. Training was conducted sequentially, using the explicit memory approach from Section 7.3.4. The class targets used 1-of-n encoding, training was conducted over the full dataset for 30 epochs with a learning rate $\alpha = 0.1$.

In the second phase, the FQI agent was then trained over 600 episodes according to Algorithm 2 (and modifications for Scenario 2). To evaluate the learned behaviour, exploration was deactivated, rendering the whole process deterministic. The system was then presented with all four input patterns. Figure 23 illustrates its response for each case and the decision process during an episode.

Since only four patterns were used without any noise, the system quickly converged to a perfect solution, always classifying the correct pattern after looking at 2 features at most.

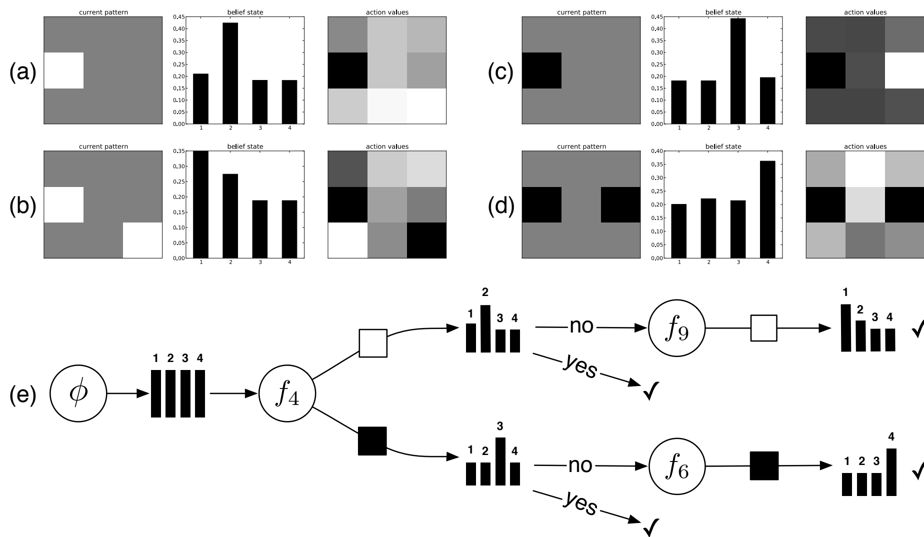


Figure 23: Decision process of the SOFS agent after training. (a)–(d) each show the already uncovered features in the explicit memory (left), the belief state histogram (middle) and the action value table for selecting the next feature (right, white indicates high values). (e) shows the decision process graphically. Initially the agent always sees the *missing* value and chooses to look at feature 4 first. If the feature is white, the classifier favours class 2, and SOFS proposes to select feature 9, should it be wrong (a). After looking at feature 9, the classifier then favours class 1 (b). If feature 4 was black, however, the classifier favours class 3 and SOFS suggests to select a *different* feature next, namely feature 6 (c). After looking at it, the classifier now favours class 4 (d).

7.5.2 Toy Example II - Cube

In this second toy example, we created an artificial dataset with an arbitrary number of features F, which can be set upon creation of the data. The idea is to have some useful information hidden in each data point while most of the features are non-informative random values. The position of the useful features are dependent on the class label, however.

This is how we created the cube dataset: Three of the features indicate x-, y-, and z-coordinates in a three-dimensional space. Each of the coordinates was randomly chosen from a Bernoulli distribution with probability $p = 0.5$ to be either 0 or 1. Then, some normally distributed noise ϵ was added to the coordinates with $\epsilon \sim \mathcal{N}(0, 0.1)$. This places each of the points around one of 8 corners of a cube, as shown in Figure 24. All other features carry no information and are initialised with a uniformly drawn value between 0 and 1. The goal is to classify each of the

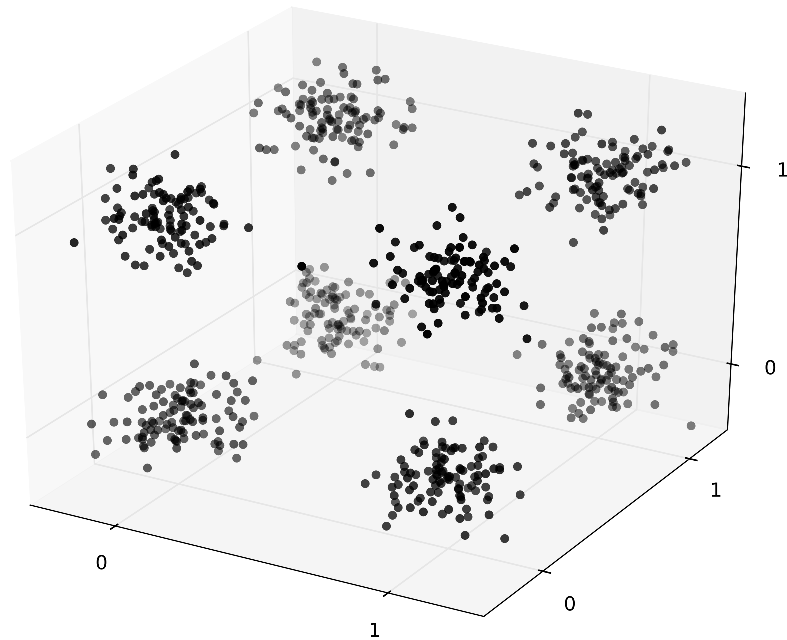


Figure 24: Visualisation of the cube toy dataset. Each data point is assigned to one of the 8 corners of a three-dimensional cube with a normally distributed noise in each dimension added. These three meaningful coordinate features are then combined with a number of non-informative random features. Furthermore, the indices of the coordinate features are different for each class.

points into its correct cube corner. While it would be an easy task for any feature selection method to isolate the three information- carrying features from the random ones, we added one extra processing step to the dataset: The three coordinate features are not at the same position in each data point, but shifted depending on their class label. The index of the x coordinate within a data point for class label c_i is $(i \mod |F|)$, y and z are positioned at $(i+1 \mod |F|)$ and $(i+2 \mod |F|)$, respectively. The modulo operator ensures that the coordinate indices are on valid positions, in case there are less than 10 features in the dataset.

This means that for class 1 with corner coordinate $(0,0,0)$, a data point would be $(x,y,z,r_1,r_2,r_3,\ldots)$, with r_i being the random features. For class 4 and corner coordinate $(0,1,1)$, the data points are defined as $(r_1,r_2,r_3,x,y,z,\ldots)$, and so on.

Conventional FS methods can only fail with this dataset (it is constructed that way). None of the features by itself is meaningful across all classes. The best they can achieve is to pick the 10 features that contain some coordinate information. While this seems to be a very unfair and artificial

Figure 25: Illustration of the coordinate feature placement for the cube dataset. The white boxes resemble random features which carry no useful information for the classification task. The black x, y, and z boxes are the informative features that describe a coordinate in three-dimensional space. The coordinate features are shifted by one position for each class. Conventional FS methods can only select features independent of the class and thus their best possible outcome is to choose the 10 features that carry useful information.

dataset, biased towards SOFS, one should keep in mind that it is quite common for features to only carry information in a certain class, or in other words, having class dependency. This experiment demonstrates that SOFS can use this class dependency to select those features during the classification process.

For classification, we use a logistic regression classifier with a soft-max output for multiple classes. We tested datasets with 5, 10, 20 and 30 random features added in addition to the 3 meaningful coordinates. Each experiment was repeated 25 times with feature costs set to $r_k^- = 0.1 \; \forall k$ and $r^+ = 1.0$ and we compare the development of the number of necessary features to access before correct classification occurs. Figure 26 shows the results over 350 episodes of training the SOFS agent. In all cases, the number of required feature quickly drops significantly below 10, the best possible outcome for conventional FS methods. Initial number of features and final number of features are displayed at the axis sides left and right respectively.

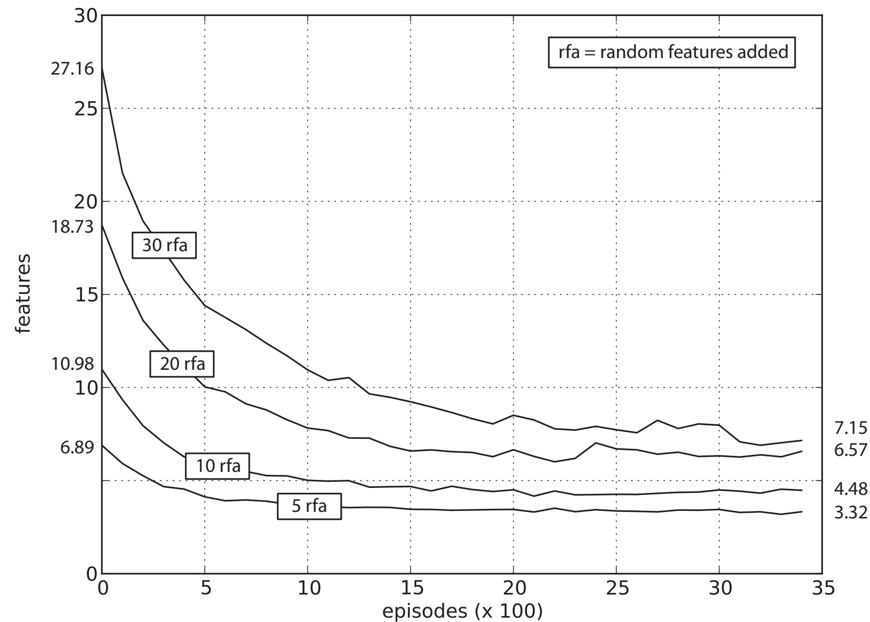


Figure 26: Results of the SOFS training process on the cube dataset. The four plots show instances of the dataset with a different number of random features added (rfa) to the 3 informative features. In all cases, SOFS learns to ignore most of the random features and mostly focuses on the informative ones. However, the more random features were added, the more difficult it is to find the 3 coordinates. Initial number of features before training and final number of features are displayed at the axis sides left and right respectively.

Comparison to Traditional Feature Selection

We also applied some conventional feature selection methods on the Cube dataset with the Machine Learning tool WEKA (Hall et al., 2009) with 10 random features added on top of the three meaningful coordinates. Two versions of the Cube dataset were converted to the WEKA-compatible .arff format. The first version named “unshifted” was created as described above, but did not contain the last processing step of shifting the feature indices depending on the class. Thus it contained the x, y and z coordinates of each point at indices 1, 2 and 3. The second “shifted” version additionally contained that last step and was in fact the same dataset used for above experiments with SOFS.

WEKA ships with a variety of feature selection (FS) algorithms. For this experiment, we first chose a Ranking based FS method, which uses an Information Gain measure to rank each feature individually (see e.g. Guyon and Elisseeff, 2003), and then selects the highest ranking features

based on a cut-off threshold. Table 2 shows the results for both the unshifted and the shifted dataset. The results are as expected: Features 1, 2 and 3 are chosen for the unshifted version, as they are the only ones that carry information (centre column). For the shifted version, the features with gain rank larger than 0.0 are features 1–10, the ones that, pooled over all classes, carry coordinate information. Comparing the information gain values to the illustration in Figure 25, we can also see that features that carry usable information more often (features in columns 3–8 carry coordinate information 3 times each), have higher information gain values, whereas Features 1 and 10 (both are only meaningful for one class) have lower gain values respectively.

Choosing the selection threshold just above 0.0 would exclude the 3 useless features 11–13, but a further reduction in the number of features (by raising the threshold even higher) would also incur a reduction in classification performance, as useful (and, in fact: required) information would be discarded.

While feature selection complexity based on ranking scale linearly with the number of features (each feature has to be evaluated only once), they do not factor in subsets of features, which may—in combination—yield a better classification rate while perhaps performing weakly on an individual basis. A second approach in feature selection is therefore to search (and evaluate) the space of feature subsets. This, of course, becomes unfeasible quickly with growing feature numbers, as the number of possible subsets grows as 2^n with n features, including the full and empty set of features. Exhaustive searches in the subset space are thus not applicable for large problems. There are many different heuristics to search the space of feature subsets, including Genetic Algorithms, Best-first hill-climbers or Greedy Stepwise approches.

We chose to add 10 additional random features to the three coordinates to make up a total of 13 features. Exhaustively searching the complete subspace ($2^{13} = 8192$ features) is still doable but already takes many hours on a current computer, as for each subset, a classifier has to be trained on the full training set.

Instead, a Greedy Stepwise search was used, starting with the empty subset, and adding features greedily until performance on the classification task decreased. The classifier used for the FS process was a simple logistic regression based on logistic model tree induction (Sumner et al., 2005). Experiments on both datasets were repeated 20 times. The result was always the same: for the unshifted dataset, the feature subset $\{1, 2, 3\}$ was chosen, and for the shifted dataset, subset $\{1, 2, \ldots, 10\}$ performed best, exactly as expected.

Table 2: Ranked Information Gain Feature Selection on Cube Data

Feature #	Gain unshifted	Gain shifted
1	1.0	0.333
2	1.0	0.546
3	1.0	0.693
4	0.0	0.867
5	0.0	0.842
6	0.0	0.841
7	0.0	0.84
8	0.0	0.643
9	0.0	0.592
10	0.0	0.318
11	0.0	0.0
12	0.0	0.0
13	0.0	0.0

7.5.3 Handwritten MNIST digit classification

In this experiment we looked at the well-known MNIST handwritten digit classification task (LeCun et al., 1998), consisting of 60,000 training and 10,000 validation examples. Each pattern is an image of 28×28 pixels of grey values in $[0, 1]$, the task is to map each image to one of the digits 0–9. We split every image into 16 non-overlapping 7×7 patches, each patch representing a feature.

We present results for a FFN as a non-sequential classifier and a RNN with Long Short Term Memory (LSTM) cells (Hochreiter and Schmidhuber, 1997) as a sequential classifier with implicit memory. The FFN was chosen because it is a well-understood simple method, widely used for classification. The RNN was chosen to investigate, how naturally sequential classifiers work with SOFS. Throughout this experiment, rewards were set to $r^+ = 1.0$ and $r_k^- = -0.1\ \forall k$.

The FFN has one hidden layer with sigmoid activation, the architecture is 784-300-10. The output layer uses soft-max activation with a 1-of-n coding. Pre-training of the classifier was executed online with a learning rate $\alpha = 0.1$ on the full training dataset. After 30 epochs of presenting all 60,000 digits to the network, the error rate on the test dataset is 1.18%, slightly better than reported in LeCun et al. (1998). However, this result is secondary, as the network acts merely as an environment for the SOFS agent. During SOFS training, each episode uses a random sample from the test dataset. Experience replay (Lin, 1992) is not used, as the LWPR function approximator is online in nature and can remember previous data. Figure 28 (left two plots) shows the development of episode lengths and returns during training of the SOFS agent. The average number of features required to correctly classify dropped from initially 7.65

Figure 27: Example of MNIST digits, randomly chosen. Each digit is 28×28 pixels in size with grey values between 0.0 and 1.0.

(random order) to 3.06 (trained SOFS). The rate of incorrectly classified images was 0.77%.

The architecture of the RNN classifier is 49-50-10 with LSTM cells in the hidden layer. The output activation function is soft-max with a 1-of-n coding. The RNN was pre-trained with Back-Propagation Trough Time (BPTT) with a learning rate of $\alpha = 0.01$ and a random order of features. The results are illustrated in Figure 28 (right two plots). The average number of required features decreases from 4.91 features (random order) to 1.99 (trained SOFS). The rate of incorrectly classified images was 1.71%.

7.5.4 Diabetes Dataset with Naive Bayes Classification

For the second experiment, we chose a more practical example from the medical field, the Pima Indians Diabetes data set (Frank and Asuncion, 2011). We also decided on a Naive Bayes classification, to demonstrate the flexibility of the proposed method in terms of classifiers. The data set consists of 768 samples with 8 features (real-valued and integer) and two target classes (diabetes, no diabetes). Pre-training with a Naive Bayes classifier resulted in 73% correct prediction. There are two interesting aspects in this dataset. Firstly, it contains missing values, which should be handled well as we already use missing values to turn classification into a sequential process. Secondly, the features represent very different attributes of the (all female) patients. Some are simple questions (e.g. age, number of times pregnant), others are more complex medical tests (e.g. plasma glucose concentration after 2h in an oral glucose tolerance test). While the MNIST experiment used uniform costs r_k^- for all features f_k, this experiment demonstrates another property of SOFS: the feature costs can be weighted, representing cheaper and more expensive features. To investigate the difference between uniform and variable feature costs, two sets of experiments were conducted: The first uses uniform costs $r_k^- = -0.1 \,\forall k$, with a final number of required features of 3.7 on average. The second variant uses variable, estimated

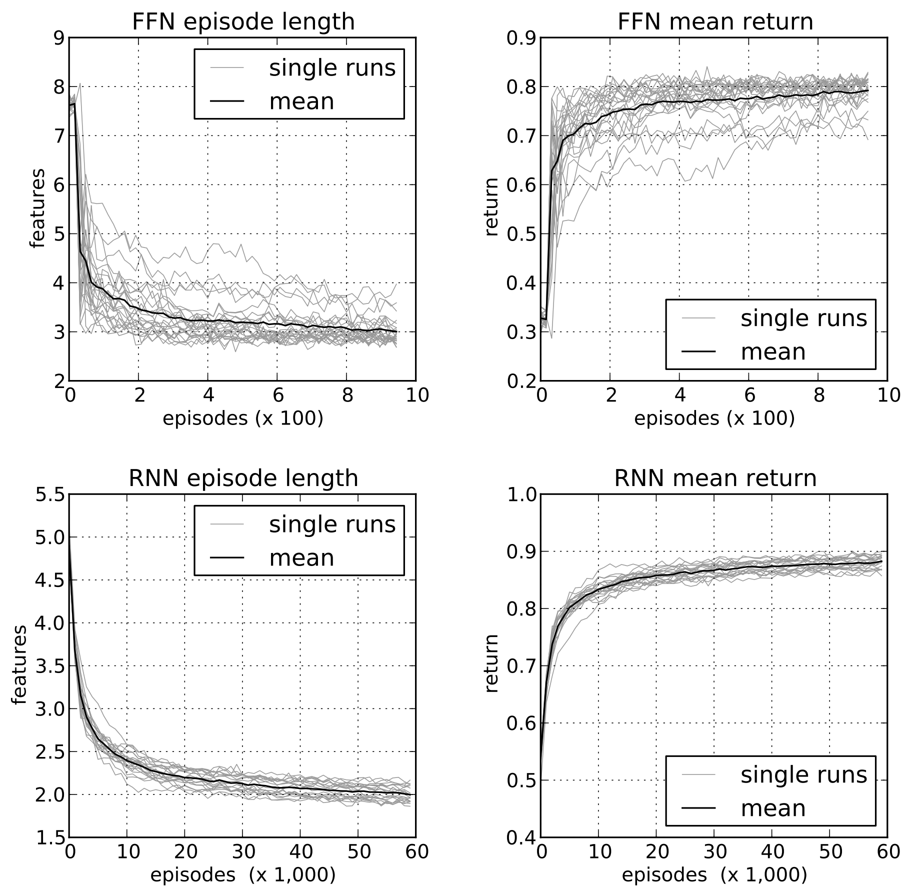


Figure 28: Results of MNIST with FFN (left two plots) and RNN (right two plots). For each classifier, mean episode length and mean return over training episodes are shown.

costs[5] shown in Table 3. Number of features *increased* from 4.99 to 5.66 on average, while the average return increased from -218 to -141. Figure 32 shows the results of both variants graphically.

7.5.5 Integrated Classification on Cube Dataset

For the last experiment, we used the SOFS+IC implementation, that learns to stop the classification process by itself. It is therefore a fully compatible classifier, while still minimising intra-sample data consump-

5 These costs represent a rough estimate of the time in minutes it takes to acquire the feature on a real patient. The estimates are based on oral communication with a local GP.

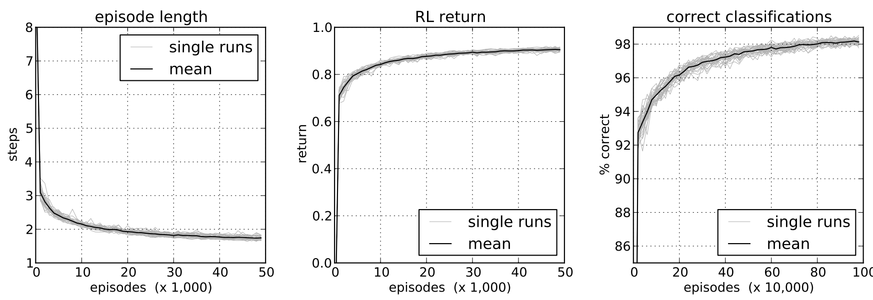


Figure 29: Results of MNIST with FFN, simultaneous training of classifier and RL (Scenario 3). The ratio of supervised to RL training was 20:1 (hence the different scales on the x-axis). Figures from left to right: episode length, mean return, correct classifications.

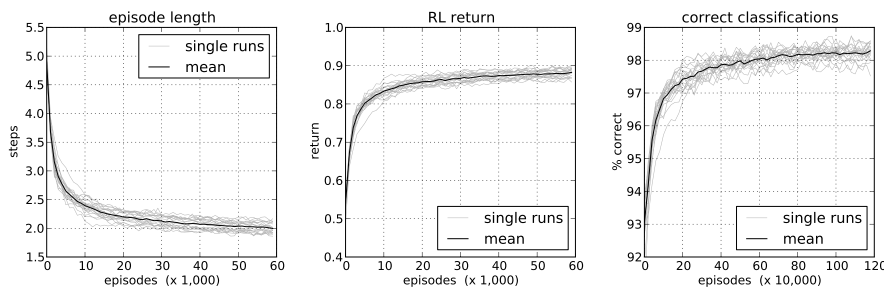


Figure 30: Results of MNIST with LSTM, simultaneous training of classifier and RL (Scenario3). The ratio of supervised to RL training was 20:1 (hence the different scales on the x-axis). Figures from left to right: episode length, mean return, correct classifications.

tion. The results are listed in Table 4, where the better number for each comparison is in bold.

We used the Cube dataset as before (Section 7.5.2) with a varying number of random features added (#rfa, first column). This time, the agent decided, after how many features it was sure enough to find the right label, without any manual intervention based on the true target. We split the dataset into 2/3 training and 1/3 verification and used the latter to get the numbers of the third column "% correct" for either method, averaged over 20 trials. The last column shows the number of features accessed for both the decision tree method and the SOFS+IC algorithm.

Most classification algorithms do not select features by themselves but rely on prior feature selection separate from the actual classification. Decision trees are an exception as they, too, look at individual features and branch out on the most informative one at each level. They are therefore ideal candidates to compare against SOFS+IC, because we can average

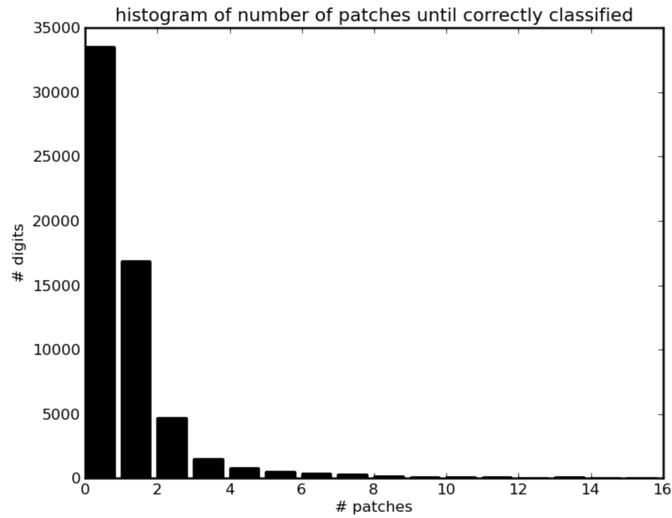


Figure 31: Histogram of the number of accessed features until correct classification occurred in the MNIST with LSTM experiment (Scenario 3). In most cases, it was sufficient to only look at a very small number of features. Less than 5% of the input samples required more than 3 features (out of 16) to be identified.

over the depth of the branches for each sample to get the number of accessed features.

7.6 DISCUSSION OF RESULTS

Early on we wanted to find out, whether the SOFS agent is in fact able to learn to select features based on a current observation or if the selected features simply improve the results on average, independent of the belief. The results of the first toy experiment delivered an answer

Table 3: Assigned feature costs for diabetes dataset in the variable-cost case.

# pregnant	2h glucose concentration	blood pressure	skin fold thickness
-1	-120	-5	-5

2h serum insulin	BMI	diabetes pedigree fct.	age
-120	-5	-60	-1

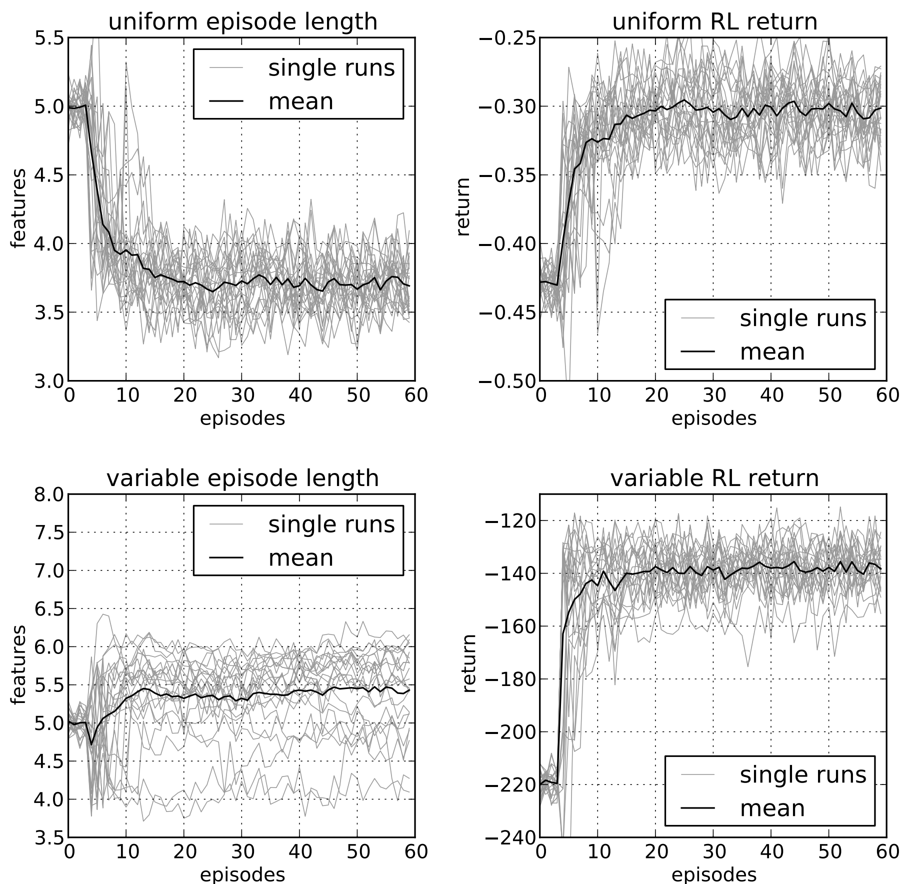


Figure 32: Results of the PIMA diabetes dataset with Naive Bayes classification. Left two figures: episode lengths and mean returns for uniform feature costs. Right two figures: episode lengths and mean returns for feature costs according to Table 3.

to that question: Figure 23 shows two cases (a) and (c) with different states, leading to the selection of feature 9 and feature 6, respectively. A first key finding is therefore, that SOFS is superior to any traditional FS method in that it can select features during a decision sequence *dependent* on the current observation.

The second toy dataset "Cube" was artificially created to demonstrate that features with class dependencies are a real issue for conventional FS methods. We did not include any comparison to other FS methods but assumed that they would optimally choose the 10 features[6] that carried useful information overall. In all cases, SOFS could beat that number

6 with the exception of the 5rfa experiment, which only has 8 features in total. All of them carry information and an optimal static FS method would have to choose all 8.

Table 4: Comparing sequential online feature selection with integrated classification (SOFS+IC) with a decision tree algorithm (C4.5). The first column indicates the number of random features added (rfa) to the three informative features. The third column presents the average error rate on a held-out test set, the last column gives the average number of accessed features over all classifications in the test set.

# rfa	classifier	% correct	mean #feat
3	C4.5	**94.9**	7.56
	SOFS	94.8	**4.17**
6	C4.5	**93.7**	8.21
	SOFS	93.4	**4.79**
10	C4.5	92.3	8.46
	SOFS	**94.3**	**4.83**
20	C4.5	94.5	8.89
	SOFS	**94.7**	**5.00**
30	C4.5	93.1	9.55
	SOFS	**93.7**	**5.03**
50	C4.5	94.2	10.34
	SOFS	**94.3**	**6.11**

and reduce the amount of required features to a significantly lower value by making the selection process dependent of the class belief.

The MNIST experiment with FFN classifier demonstrates a significant reduction of data consumption in two ways. Firstly, by making the decision process sequential, which enables the classifier to make decisions before all features have been looked at. This step alone reduces the average number of required features from all 16 features down to 7.65 (a reduction to 48%), and indicates that there is in fact a lot of redundancy in the MNIST images. Secondly, consumption is reduced further by learning the dependency of current belief and next feature, instead of accessing them in random order. After training the SOFS agent, data consumption decreases to 3.06 on average, 19% of the full data.

It is important to note that the stated error rates (1.18% for static and 0.77% for sequential classification) cannot be compared directly, because of the very different nature of the sequential approach. Sequential classification replaces the conventional error rates as performance measure based on the binary success of each sample (classified / not classified) with a scalar value (how many features until classified). In order to compare both classification methods, we would have to additionally learn when to stop the decision process, without using the class label. This could be achieved with a confidence threshold (e.g. if max(belief) reaches a certain value (Norouzi et al., 2010) or by explicitly learning when to stop with either supervised or RL methods (Dulac-Arnold et al., 2011b). In this paper, we focused on the RL feature selection process

with existing classifiers rather than the performance of sequential classifiers. This issue will be addressed in a future publication.

Another aspect we investigated was the use of RNNs as naturally sequential classifiers. Where static classifiers still need to look at a full input (at least in terms of dimension, even though most of the pattern is filled with missing values), RNNs can make use of their intrinsic memory and achieve similar results with significantly fewer nodes in input and hidden layer and therefore even less data processing. They also converge with lower variance and reduce data consumption to a mere 12% on the MNIST task.

The Pima diabetes data set illustrates the use of variable feature costs, a variant that is naturally supported in our framework. The left two plots in Figure 32 show the development of episode length (i.e. number of selected features until correct classification) and mean return of the uniform cost experiment. As expected, episode lengths decrease with increasing returns, as the only objective for the agent is: *select those features first, that lead to correct classification.* However, if the reward scheme is changed (right two plots in Figure 32), we witness a *growth* of episode lengths in most of the 25 trials and on average. Still, all trials increase their returns (rightmost plot), which indicates that the agent does indeed learn and improve its performance. Comparing the final return average of -141 and the worst final return of -160 to the individual costs of Table 3, it becomes clear that in all runs, only one of the three most expensive features (number 2, 5 and 7) was selected. This behaviour was caused by the different objective: *minimise the overall costs associated with the features.* In other words, it is okay to select many features, as long as they are cheap.

Lastly, the results for SOFS+IC show that a reformulation of the problem allowed us to include classification as part of the decision process. SOFS+IC reached comparable results with a standard C4.5 decision tree algorithm for small numbers of added random features (#rfa), and slightly outperformed the decision tree algorithm for higher #rfa in terms of correct classification on the validation set. Looking at the average number of features accessed per trial, SOFS+IC shows a dramatic reduction in numbers, almost half of the features required by the decision tree algorithm.

8 CONTEXT LEARNING

8.1 INTRODUCTION

This chapter discusses a novel approach to sequential classification, using Reinforcement Learning to build up context during sequence traversal. At each time step, a classifier only considers the current input and its context. The reinforcement learning agent is rewarded based on the performance of the classifier, improving the context-building strategy over time.

Before we get into implementation details, the following sections discuss the fundamental idea and give some examples.

8.1.1 Spatial Context — Some Introductory Thoughts

To understand the concept behind context learning, let us discuss the common problem of linear separability in a different light on a very basic and fundamental level.

Assuming a fictitious collection of two different kinds of objects, that we will call x and o, and a binary classification task, we would like to find a rule that separates every x from every o. "Separation" implies that these objects share an abstract space in which they are embedded, and each object's location in this space is defined by projections of its true form onto the axes of that space. A dimension could be the object's size, its colour, the time it existed, its placement in any of the three spatial dimensions or any other arbitrary measurement that captures a property of these objects.

What does it mean, then, to separate one group of objects from another? With infinitely many ways of describing an object, there are infinitely many possible rules to separate one group from another[1]. Still, asking to separate all x objects from o objects seems fairly straight-forward: "Put the ones that look like a cross on one pile, and the ones that look

1 There are only a finite number of possible partitions into two groups: n objects can be grouped into two indistinguishable groups in $2^n - 1$ different ways. Yet the number of possibilities to describe these partitions is infinite, assuming the number of object properties is infinite.

like a circle on the other pile." This is because they have some obvious commonalities amongst each group, which differs between groups. But the term "obvious" is not formally defined and therefore arbitrary. Perhaps someone else finds it more obvious that some of the objects were created a year ago (including some *x*s and some *o*s), while others were created yesterday. This person would group the objects in different piles. Who did it right?

We need to introduce another objective to the problem: We would like to find a *simple* rule, that separates every x from every o. And adhering to Occam's Razor (Blumer et al., 1987) we could define "simple" as being a straight line (or hyper-plane for high-dimensional space). This brings us to the issue of linear separability.

It is well-known that with limited amount of information available (finite number of dimensions), linear separation is not always possible. See Figure 33 (a) for example. Separating the objects denoted as x from the objects o is not possible with a straight line. With no further information, one would have to find a very complex (non-linear) rule (blue dotted line) to separate the letters from each other so that x and o are separated. A linear partition (green solid line) is plausible, but it does not provide the groups we would expect.

This problem is due to the lack of information about these objects and stems from the fact that everything we know about any object must necessarily be an *observation*, an image of that object, and not its true form. Because of that, we never have complete information about any objects, but only interpreted, incomplete projections. It is these projections that are not necessarily separable with an easy rule (straight line).

The situation changes if we allow for some context to be added to the observations. We define the context to be some additional information that was not observed, yet is still part of the consideration for classification (we will explain further below how this is possible). In a way, the context is similar to *closures* in some programming languages (like Lisp). Closures keep the extra state of non-local variables, the context, in which functions are being evaluated. In Figure 33 (b) (bottom graphic), the context is plotted on the y-axis of the graph. In this case, the context reflects the shape of the object. Looking at the observations in the right context, the objects now look different enough to be separated easily, as shown with the red solid line in the figure.

This seems like cheating, because we basically just provided the desired solution and called it "context", but it does not always have to be that simple. We could have looked at the amount of ink per pixel, or counted the corners of each symbol. Each of these additional pieces of information would have helped to find a linear separation. There are also oth-

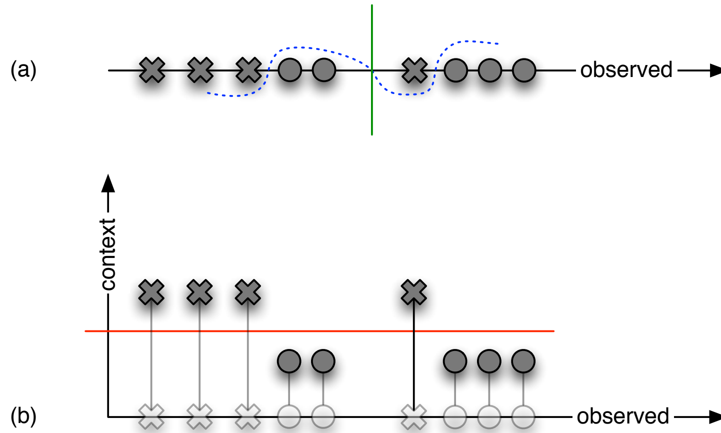


Figure 33: Illustrating spatial context on a linear separability problem. a) Observations of x and o symbols are plotted on the x-axis (grey symbols). There is no easy rule (straight line) that can separate the x and o symbols based on the observations. Instead, a complex rule (blue dotted line) is required to solve the binary classification task. The green line does not partition the objects in an acceptable way. b) Considering the observations in the right context (context plotted on y-axis), the symbols can now be separated linearly (red line).

ers, that would not have helped. The colour for example, or the person drawing the symbols were identical across all objects. There is no way of knowing the correct context and finding it is often a trial-and-error process. In fact, in what follows, we will *build* a context of observations sequentially, without actually knowing what exactly this context state describes. That is where Reinforcement Learning will come into play. The agent will guess an arbitrary contextual state, see how well classification works within the context, and adapt the context appropriately. In all but the most trivial examples, this contextual state is a black box and cannot be understood explicitly. This is comparable to the parameter values of many parametric machine learning algorithms, that do not hold any explicit meaning outside of the algorithm.

Related to this notion of context is the concept of Perceptual Aliasing (Whitehead and Ballard, 1991), a term that describes an indistinguishable internal state of a RL agent that maps to multiple real-world states. Such aliased states often lead to noisy credit assignment as the seemingly unique state is really a conglomerate of multiple outside states, usually requiring a different action and yielding different reward. Approaches to tackle Perceptual Aliasing include algorithms to build higher-order memory context (Ring, 1993), an idea that can also be found in the (Noisy) Utile Suffix Memory algorithm (McCallum, 1995; Shani and Brafman, 2004), Perceptual Distinctions (Chrisman, 1992), the

CHILD algorithm (Ring, 1997) as well as more traditional approaches like fixed-size history windows or eligibility traces.

These algorithms focus on the Perceptual Aliasing problem within a Reinforcement Learning framework, by splitting aliased states or enhancing or replacing state information via historical data or preditive models. In this chapter, we take a different approach and look at context in a more general angle outside the RL framework, using RL only to create appropriate context to augment the input into a supervised classification algorithm.

8.1.2 Temporal Context

The last section was a very abstract and general introduction to the concept of context. Now we want to look at a more specific example: temporal context.

Where does the context appear in temporal sequence learning problems? In Section 8.1.1, the context was mere additional information, that was not covered in the observation. For time series data, the context acts as the memory for what has been seen so far. The first data sample appears in an empty context, while subsequent time steps happen in contexts that each summarise the past experience up to the current time. Also see the remarks about Sequence Learning in Section 4.7.

The same notion of observation within a context can be found in time series prediction (Connor et al., 1994; Sapankevych and Sankar, 2009) or sequence learning (Maes et al., 2007; Graves, 2008; Sun and Giles, 2001). Recurrent neural networks for example store temporal context in their hidden recurrent units. This approach is not unlike Context Learning, however the learning algorithms differ significantly. Training RNNs with any kind of back-propagation algorithm suffers from the vanishing gradient problem, and therefore imposes limitations of the length of sequence (here corridor length) the network can learn. Methods like Hierarchical Enforced SubPopulations (H-ESP) (Gomez and Schmidhuber, 2005) do not have these limitations, but the computational complexity is much higher, due to the many iterations required by the genetic algorithm to find a good solution in the gene pool. Here, we present a method that is much less computationally expensive yet will find general solutions not dependent on corridor length.

Here we consider the problem domain called *T-Maze* (Bakker et al., 2002; Gomez and Schmidhuber, 2005; Wierstra et al., 2007). An agent walks through an east-facing corridor of length n, one step at a time. At some

point t, the agent receives a "road sign" signal[2] if it should go north or south at the end of the corridor. No further information is given. At the end of the corridor is a T-junction, leading north and south. Based on the road sign perceived earlier, the agent has to pick the correct direction.

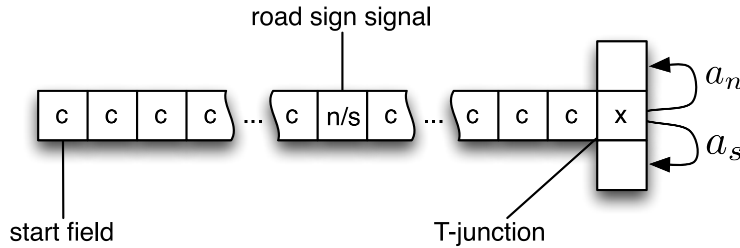


Figure 34: T-Maze problem. An agent steps through a corridor (c) from west to east and receives a road sign signal (n or s) at some point in the corridor. At the T-junction (x) it has to move north (a_n) or south (a_s), depending on the road sign signal received earlier. This problem is a well-known partially observable Markov Decision Process (POMDP) and used as a benchmark for deep memory policies.

We define the following symbols for the states of the T-Maze task: An empty corridor field is denoted c, a corridor field with road sign signal is written n for north or s for south, and the corridor T-junction field is denoted as x. Assuming that the agent moves east through the corridor by default[3], the only required non-standard actions are a_n and a_s for moving north or south, respectively. A few typical sequences with the correct action at the end could then look like these:

$$\begin{aligned} ccncccccx &\rightarrow a_n \\ cccccsccx &\rightarrow a_s \\ ncccx &\rightarrow a_n \\ ccccsx &\rightarrow a_s \\ cccscccccx &\rightarrow a_s \end{aligned} \tag{8.1}$$

It should become clear from these examples, that the T-Maze problem in this formulation is a (binary) sequence classification problem. Those sequences containing an n fall in the class a_n and those with a s fall in a_s.

2 The original T-Maze task is defined with a slightly easier task, where the agent always receives the "road sign" signal at the start rather than at an unknown point in the corridor.

3 Adding a separate action a_e to move east does not make the task more difficult, but distracts from the actual problem. Therefore, it is left out in this formulation, without loss of generality.

Recent solutions (Bakker et al., 2002; Gomez and Schmidhuber, 2005; Wierstra et al., 2007) to the T-Maze problem are all formulated as reinforcement learning tasks in a POMDP setting, using recurrent policies to tackle the memory problem. While solutions for short corridors can usually be found easily, the problem difficulty increases with longer corridors, because the recurrent policy (in above publications implemented as LSTM network, refer to Section 4.5.2 for details) must implicitly remember longer and longer sequences to find the useful bit of information (the road sign signal) in their past experience (Bakker et al., 2002; Wierstra and Schmidhuber, 2007).

Here, we will take a very different approach to solving the T-Maze problem, one that starts with a seemingly very simple question leading us back to the notion of context: Why can't we treat each of the symbols in a consistent way? Why are there different actions required when encountering the same symbol? In one sequence, an x requires a_n, and in another, an x requires a_s to satisfy the task. Just like with the x in the x/o example above (Section 8.1.1), the observation was incomplete and lacking context. Before, the context was of spatial nature, given as a separate dimension. In the T-Maze task, the context of x is temporal, summarising the past sequence leading up to x. Had the agent access to this context, the classification would become trivial, because the T-junctions would not look the same. They would become distinguishable, something like x_n and x_s, and the two different states would always require the same actions: $x_n \rightarrow a_n$ and $x_s \rightarrow a_s$, where previous time steps would not matter anymore.

Aside from the fact that we have not yet discussed how to get to this context, this approach makes the T-Maze a non-sequential problem, because the input + context at each decision step become independent of previous inputs. The context acts as a wrapper to cover all temporal dependencies, just like the recurrent policy did in e.g. (Bakker et al., 2002). One of the problems, that recurrent neural networks often suffer from, is that of the vanishing gradient (Hochreiter, 1998). The advantage of using a context is that longer corridors won't make the problem more complex and as such do not suffer from vanishing gradients. The solution is in fact independent on the corridor length, which makes context learning a very promising and attractive alternative to recurrent policies for sequence learning tasks.

8.1.3 How to Learn the Context

From above examples, the context seems to be some magical black box that contains enough information to make the solution trivial. Obviously, learning the context is not a supervised problem, or else we would

have to know the correct context for each sample, which is almost as good as knowing the correct target.

It is furthermore unclear how much and what kind of information the context should contain to solve a problem in general. In the T-Maze task, the question of context size at least is trivial: One bit is enough information to solve the decision at the T-junction: go north or south.

Our proposal for getting the right context is to use Reinforcement Learning to build the context while going through a sequence and to reward those sequences that were correctly classified (they must have had a good context building policy). Initially, the context will be random, and correct classification will be sparse at best, but over time, both the classifier and the reinforcement learner will co-improve in parallel and return better solutions.

Building a context as a RL problem can be done in different ways. One approach is to have an explicit memory, a bit vector of certain length, and actions that allow editing the values of this vector. This idea reminds of the basic principles of a Turing Machine, and is in fact an interesting extension to the simpler context building strategy that we will use here: Instead of editing a vector, we define a finite number of disjunct context states that the agent can switch between. One could imagine concatenating a representation of the context state to the observation, but another possibility (one we chose here) is to use a different classifier for each context state.

In the T-Maze example, the agent would walk trough the corridor and decide with each input, what context state it wants to change to. Each context state then uses a separate classifier to classify the current input. Having two classifiers for x, one for the case where the context is to go north, and one for going south, is equivalent to having a single classifier but augmented input states $(x, 0)$ and $(x, 1)$, where the second value in the tuple represents the 1-bit context. Both make the situation distinguishable and the action choice trivial.

While spatial and temporal context learning are not that different from each other, we will focus on temporal context learning in the next sections. It is important to point out, however, that the basic idea can be applied to spatial context as well. Spatial problems would use a similar approach to Sequential Feature Selection (Chapter 7) and go through the full sample in some pre-defined order, to incrementally build up some context. SOFS could even be combined with context learning, where SOFS learns the order of features to look at while building a good context for the final classification. Below, we will focus on naturally sequential input patterns, define context learning formally and look at its properties more closely.

8.2 CONTEXT LEARNING FRAMEWORK

In the following Section 8.2.1, context learning will be formally introduced for discrete states and actions. Section 8.2.2 will then point out some similarities and key differences to Hidden Markov models (HMM). Section 8.2.3 gives an analogy of context learning to Deterministic Finite Automatons (DFA) and regular expressions.

8.2.1 Discrete State Context Learning

Context Learning is a very general concept to augment observations with contextual information, disambiguating inputs and making them easier to predict or classify. The contextual information is gained through a RL component that chooses a new context based on the current input and context and learns over time to create useful context that helps the underlying supervised prediction/classification task.

Several variants of context learning are possible and we will discuss some alternatives in Section 10.3. In this specific implementation of context learning, the following assumptions are made: We assume a classification task with a training set of n input sequences with lengths T_n from a finite alphabet $\mathcal{X}$ of possible inputs and output sequences from a finite alphabet $\mathcal{O}$. Each data sample has an associated same-length vector of importance factors i_t^n. The training data I is:

$$I = \left\{ \left((x_1^n, x_2^n, \ldots, x_t^n), (o_1^n, o_2^n, \ldots, o_t^n), (i_1^n, i_2^n, \ldots, i_t^n) \right) \right\} \tag{8.2}$$

with $x_t^n \in \mathcal{X}, o_t^n \in \mathcal{O}, i_t^n \in \mathbb{R}$ and $0 \leq i_t^n \leq 1 \; \forall t, n$.

A set of contexts $\mathcal{C}$ is defined with $|\mathcal{C}|$ elements $K_1 \ldots K_{|\mathcal{C}|}$, each representing a copy of the classifier K with different parameter sets θ_n. Classifiers $K_n(x;\ \theta_n)$ are functional mappings from the input alphabet space $\mathcal{X}$ to the output alphabet space $\mathcal{O}$:

$$K_n : \mathcal{X} \mapsto \mathcal{O}, \quad x_t^n \to o_t^n \tag{8.3}$$

Note: The classifiers do not map the full sequence x^n to the sequence o^n but only individual elements of those sequences.

The RL problem of which context state to choose at each given point in time can be defined as a Markov Decision Process (MDP) with the 4-tuple (S, A, $\mathcal{P}$, $\mathcal{R}$): S is the state space, comprised as the cross product of context states with the input alphabet: $S = \mathcal{C} \times \mathcal{X}$, therefore each RL state consists of a tuple of classifier and input (K_t, x_t^n). The action

space of the MDP is identical to the space of context states, $A = \mathcal{C}$, thus choosing an action a_n is identical to choosing a classifier K_n. Transition probabilities $\mathcal{P}$ of moving from state s_t to s_{t+1} with action a_t are defined by the deterministic transition function $s_{t+1} = T(s_t, a_t) = (a_t, x_{t+1})$. The reward function $\mathcal{R}$ returns the following rewards when transitioning to another state s_t with action a_t:

$$r_t = \begin{cases} i_t^n & \text{if } a_{t-1}(x_t^n) = o_t^n \\ 0 & \text{else} \end{cases} \tag{8.4}$$

where $a_{t-1}(x_t^n)$ is the classification of the current input element x_t^n with the classifier chosen at the previous time step $t-1$ (note that the action space $\mathcal{A}$ is the same as the space of contexts $\mathcal{C}$ consisting of classifiers K_n).

Finally, we define the initial action $a_0 = K_1$ (with any arbitrary classifier, here the first one) and start the MDP at time $t = 1$, to get a well-defined reward r_1 for the first iteration.

This definition allows for rules of the following structure to be learned: "If in context c_1 and input x is observed, switch to context c_2".

The classifiers and the RL agent are trained concurrently, as illustrated in Algorithm 4. Training occurs in lines 14 and 15 online. Only the current classifier (chosen by the last action) is trained with the current sample. Alternatively, the training samples (associated with the right classifier) can be stored and classifier and agent are batch-trained after a number of episodes. The importance value at each time step affects both classifier and agent. During training of the classifier, the current importance factor is multiplied with the error. For importance of 0, no error signal reaches the parameters of the classifier and it remains unchanged. The importance factor is also the reward for correct classification at each time step. Only positive importance (usually with value 1.0) influences the policy during training. Figure 35 shows a graphical representation of the full system with supervised component above the dashed line and reinforcement learning component below.

To ensure deterministic behaviour throughout one episode, we use the discrete variation of state-dependent Exploration, introduced in Chapter 6.3.6. A single context switch can affect all future context decisions and therefore the end results. It is important to introduce no randomness during an episode, but instead explore between episodes. Discrete value-based SDE as exploration technique has lead to the best results for discrete context learning.

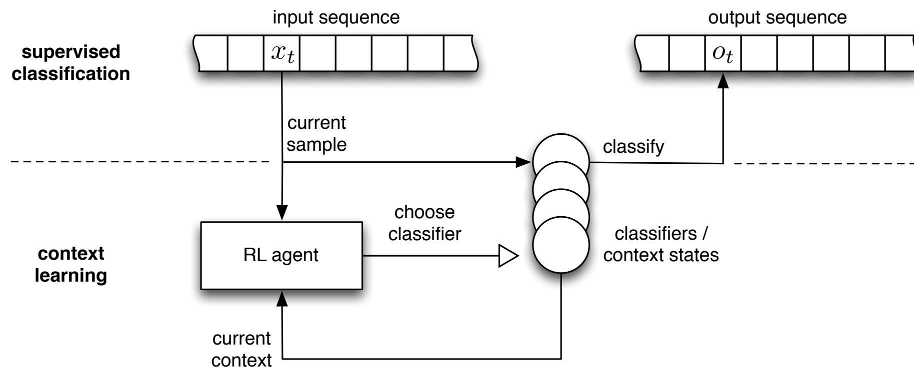


Figure 35: Overview of the discrete state implementation of Context Learning. The part above the dashed line is the supervised classifier. Each time step is classified individually. This, of course, would not work without any context. Below the dashed line is the reinforcement learning component. The agent receives the current input and the current context and chooses a new context (i.e. classifier). That classifier is then used to classify the sample.

8.2.2 Differentiation to Hidden Markov Models

Hidden Markov Models (HMM) are stochastic models for time series recognition. Despite their seemingly very similar function and application, there are some key differences to the idea of context learning, which we will point out here. HMMs are Markov chains where the internal states of the model are considered hidden, while the outputs, which depend on the states, are visible. The model has a state transition probability distribution that defines how transitions from one hidden state to another occur. Further, each state has output probabilities assigned for each of the output tokens. The outputs are assumed to be drawn from the current state's output probability distribution, before the new state is drawn from the state probability distribution. HMMs can be efficiently trained by an algorithm called Baum-Welch (Baum et al., 1970) after its creators Leonard Baum and Lloyd Welch. It calculates the maximum likelihood of state transition and output emission probability distributions given only a set of emissions (output sequences).

HMMs are Markov chains for which the states are only partially visible, but transitions to new states only depend on the previous state. In contrast, Context Learning assumes an underlying Markov Decision Process, where the transition to a new state also depends on the input and where outputs are not drawn from a probability distribution but calculated with (usually more complex) classifiers, again based on the input. Figure 36 illustrates this relationship graphically.

Algorithm 4 Context Learning for Discrete Inputs

Require: input/output/importance sequence triples I, agent A, classifiers $K_c \in \mathcal{C}$
1: **repeat**
2: choose $(x, o, i) \in I$ randomly
3: $a_0 \leftarrow K_1$
4: $s_1 \leftarrow (K_1, x_1)$
5: **for** $t = 1$ to T_n **do**
6: $K = a_{t-1}$
7: **if** $K(x_t) = o_t$ **then**
8: $r_t \leftarrow i_t$
9: **else**
10: $r_t \leftarrow 0$
11: **end if**
12: $a_t \leftarrow A(s_t)$
13: $s_{t+1} \leftarrow (a_t, x_{t+1})$
14: train K with (x_t, o_t, i_t)
15: train A with (s_t, a_t, r_t, s_{t+1})
16: **end for**
17: **until** convergence

8.2.3 Relation to Finite State Machines and Regular Expressions

It turns out that the context learning framework as described above is a generalisation of a well-known construct from the field of theoretical computer science: A deterministic finite automaton (DFA), also known as a finite state machine. DFAs (and likewise their non-deterministic equivalents, NFAs) are proven to accept a certain class of formal languages, namely *regular languages*. DFAs are therefore closely related to regular expressions, that are shown to have the expressive power of producing (or accepting) regular languages. Because context learning is a super-class of DFAs, we now have an indication (a lower bound) of the problems that context learning ought to be able to solve.

Below, we will prove that a DFA is a special case of discrete context learning where the classifier is strictly binary and the sequence ends with a terminal symbol (like in the T-Maze task, where all sequences ended with a x). We also require a sequence classification task, where the importance sequence is 0 for all but the last element.

The definition of a DFA goes as follows: Q is a finite set of states and Σ is a finite alphabet of symbols. δ is a transition function $\delta : Q \times \Sigma \mapsto Q$. $q_0 \in Q$ is the initial state and $F \subseteq Q$ is a subset of Q of accepting (or *final*) states. A DFA is then fully defined as the 5-tuple $(Q, \Sigma, \delta, q_0, F)$.

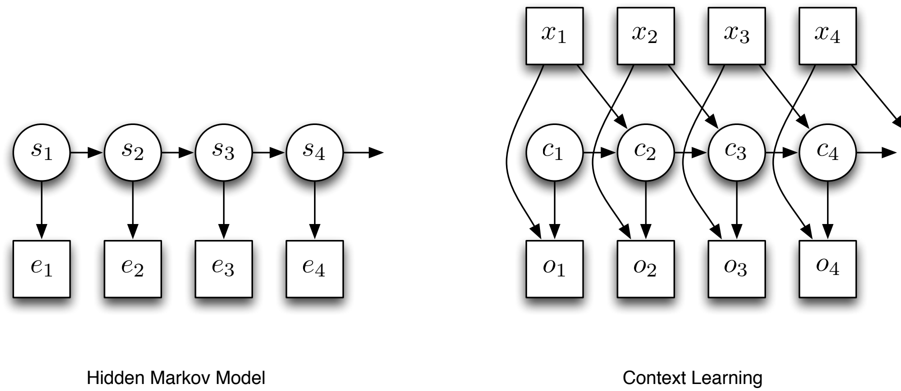


Figure 36: Hidden Markov Model (left) vs. Context Learning (right), an arrow marks a dependency from the source. HMMs are Markov chains where the state transition probability depends only on the previous state. Context Learning contains a Markov Decision Process at its core, and state transitions depend on both the previous state and the input. Another difference is a simple probability distribution over emissions for each state in the case of HMMs, and a potentially more complex classifier with its output depending again on the input.

After training the agent (with exploration) to convergence, the agent contains a final policy that maps states to actions. In value-based RL, this is achieved with a value table, where each (s, a)-tuple has a certain value assigned to it. After training has completed, the agent is switched to a fully greedy policy, disabling any exploration and only exploiting the learned behaviour. Therefore, the policy π becomes a deterministic function $\pi(s) = max_a Q(s, a)$ returning the action with the highest value for each state. Applying the following substitutions

$$\begin{aligned} Q &\longleftarrow \mathcal{C} \\ \Sigma &\longleftarrow \mathcal{X} \\ \delta &\longleftarrow \pi \\ q_0 &\longleftarrow K_1 \\ F &\longleftarrow \Big\{ K \mid K(x_{T_n}) = 1 \Big\} \end{aligned}$$

we can create a one to one mapping from our previous definition of context learning (with slight limitations, i.e. binary classifier and terminal symbol) to a finite state machine. Policy π (without exploration) maps deterministically from states to actions. The context learning state space was defined as $\mathcal{C} \times \mathcal{X}$ which becomes $Q \times \Sigma$, and the action space is $\mathcal{C}$ which is equivalent to Q. Therefore, the new δ maps from $Q \times \Sigma$ to Q, just as required. After training all classifiers, we can query them with the

terminal symbol. All classifiers that return 1 (chosen arbitrarily, without loss of generality) are "accepting" states and members of F.

The reason for the limitations of a terminal symbol and a binary classifier are there because the context learning framework as described above is more powerful and represents a generalisation of DFAs. Allowing multi-class classification would introduce final states other than "accepting" and "rejecting" the input word. And having sequences end on non-terminal symbols would allow states to change their mind of being accepting on the fly, based on the last seen symbol. If these variants indeed offer richer language representations than regular languages remains to be answered through further investigation. Knowing that DFAs are a special case of context learning at least reveals the class of problems that can be solved with context learning, namely regular languages.

8.3 EXPERIMENTS

8.3.1 T-Maze

The first experiment is the T-Maze task, which was introduced as an example in Section 8.1.2. We will describe the experiment setup again briefly, together with the algorithms and parameters used, and then list the results.

Experiment Setup

The T-Maze task describes a sequence learning problem that can be interpreted as an agent walking east along a corridor of length l (see Figure 34). At some point along the way it receives a road sign signal indicating either *north* or *south*. At the end of the corridor is a T-junction where the agent has to make a choice of either going north or south. If it chooses the same direction received as the road sign signal, it receives a positive reward of $+1$, else the reward is 0.

Formally, the problem describes a dataset of n sequences of (possibly) different lengths T_i with $2 \leq i \leq n$ over an alphabet $\{c, n, s, x\}$. Each sequence must end with x and contain exactly one instance of either n or s as a non-terminal symbol. The remaining positions are filled with c. Each sequence is associated with a label from $\mathcal{C} = \{n, s\}$. Since we already know that the class of problems solvable by context learning are identical with the problems of formal regular languages, we can give a regular expression in POSIX notation describing the class of sequences as "`c*[ns]c*x`", with $*$ being the *Kleene star* operator, indicating 0 or

more repetitions of the previous symbol, and the square brackets [·] represent a set, wherefrom one of the containing symbols is chosen. Example sequences of this class are shown in Eqn. (8.1) on page 109.

The idea behind this problem scenario is that the agent requires a memory and must remember the signal until the end of the corridor. Traditional methods of solving such deep memory tasks involve recurrent neural networks that are known to store information over a long period until it is needed. The problem for such methods becomes more difficult, the longer the sequence is. In contrast, context learning removes the sequential dependencies and generalises over the corridor length.

Context learning is comprised of two components: a set of supervised classifiers (one for each context state) and the reinforcement learning agent, choosing the context state based on last state and current input (refer to Figure 35 for a visual representation). For the classifiers, we used logistic regression, the reinforcement learning algorithm is a table-based $Q(\lambda)$ algorithm, (e.g. Sutton and Barto, 1998), with discrete state-dependent exploration (Section 6.3.6). The RL agent uses the following parameters throughout all experiments: $\alpha = 0.5, \lambda = 0.5, \gamma = 0.9$ (see Sutton and Barto (1998) for details on these parameters) and initial exploration $\epsilon_0 = 0.5$, decaying to $\epsilon_T = 10^{-4}$ over the length of each episode. For all experiments, the size n of the training dataset was 100, and a separate test dataset with which the algorithm was evaluated contained another 100 independent corridor samples.

The first set of experiments tests Context Learning on the T-Maze task with varying corridor lengths of 50, 200, and 1000 steps before the T-junction. The context size for these experiments is fixed to two possible states (1 bit) in this experiment. Figure 37 shows the results.

The second experiment then investigates the effect of varying context size with a fixed corridor length of 20 steps. Context sizes of 2, 3, 5, 10, 15, 20, and 50 states were compared. Figures 38 and 39 illustrate the results.

All experiments were repeated 30 times and the mean values over all runs are presented.

Results

Context Learning does not have to back-propagate the reinforcement signal through time (i.e., back through the corridor) as recurrent networks do (Wierstra et al., 2007), therefore we expect to see similar convergence times for all three sets independent of corridor length. Indeed, Figure 37 demonstrates this nicely. All three curves (representing corridor lengths of 50, 200 and 1000) exhibit the same convergence rates. Convergence

is achieved between 200 and 250 iterations through the training dataset, which contains 100 random corridor instances. Thus, the algorithm requires 20,000–25,000 attempts of the T-Maze to learn the task. Compared to Wierstra et al. (2007); Gomez and Schmidhuber (2005); Bakker et al. (2002), that each need millions of iterations to learn a policy for corridor lengths of up to 100 steps, it shows that Context Learning vastly outperforms these methods on the T-Maze benchmark. Not only does Context Learning requires two magnitudes less evaluations to learn the task, it does so independent of the corridor length. With the exception of Gomez and Schmidhuber (2005), which similarly learns a policy independent of time, none of the methods was reported to be trained successfully for a corridor length of more than 100.

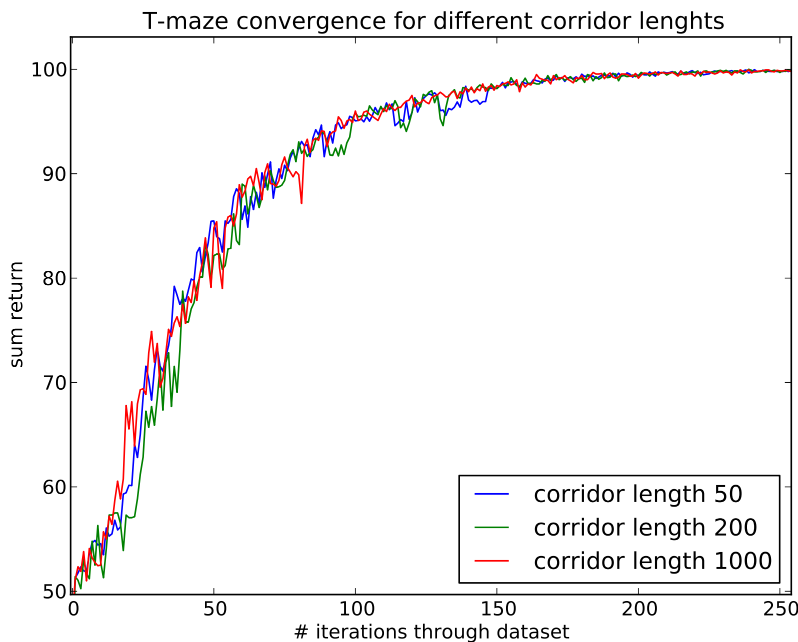


Figure 37: T-Maze results for varying corridor lengths. The three curves represent the mean return over 30 experiments for corridors of lengths 50, 200, and 1000. All three plots converge equally fast, showing that Context Learning solves the task independent of corridor length and does not have to back-propagate the reward signal back through time, as for example recurrent neural networks do (Wierstra et al., 2007).

As mentioned before, the T-Maze task can be solved with the minimum amount of context: One bit (two context states) is enough to store the necessary information at the road sign, retrieved at the T-junction to make the decision where to go. Still, the question remains how large to choose the context size for any given problem, and if bigger context

sizes have adverse effects on convergence. The next set of experiments investigated this issue. For a corridor length of 20, the context size was varied. Experiments with 2, 3, 5, 10, 15, 20, and 50 context states were conducted, 30 independent repetitions for each context size. The mean values over these 30 runs are reported in Figures 38 and the variances in 39.

On first sight, it seems that convergence is not affected by different context sizes, as all curves in Figure 38 converge nicely with the same rate. Looking at the plots more closely, however, one can see that the runs with less context states seem to converge smoother than the ones with many states. In particular, the black plot (50 context states) looks a lot more erratic. Since these plots resemble the mean over 30 independent runs, we plotted the variance of the experiments in Figure 39, top graph. In this plot, it becomes quite apparent, that there seems to be a significant difference between the number of context states, all other variables kept constant. Low numbers of states have high variance at early stages but converge quickly and variance fluctuations reduce over time, whereas the experiments with the highest numbers of states start off with lower variance fluctuations but keep a steady variation even in the late phase of the experiments. The bottom plot in Figure 39 shows this effect even more clearly, where the cumulative variance is plotted, each point in the graph representing the total sum of all variance values up to this point. Again, the variance of runs with low numbers of context states increases quickly, but reaches a plateau soon, while more context states show steady growth. In particular, the two experiments with 20 (yellow) and 50 (black) states do not seem to reach a plateau at all within 200 steps. What does this mean in terms of the convergence of the experiments? It appears that the closer one gets to the minimum number of context states required to solve the problem (here: two), the harder it is initially to get the right solution. There is less room for any errors, and no redundancies (i.e., multiple possible solutions) therefore the variance is high initially. Once the problem is solved, variance drops to almost zero and all repetitions converge equally. Increasing the number of context states means to add more flexibility to the policy controller. The problem can now be solved in many different ways, similar to an over-determined system of linear equations. There are more degrees of freedom to solve the problem, and finding an initial solution is easy, hence the variation in reward is lower initially. However, many of these initial solutions may not be perfect and could end in local minima, from which there is no way to recover. Therefore, for large numbers of context states, some of the trials may lead to sub-optimal solutions and not converge perfectly, and the variance over time remains high. This points to a known problem of many machine learning techniques: Meta-parameter optimisation. Just like neural networks have a tunable

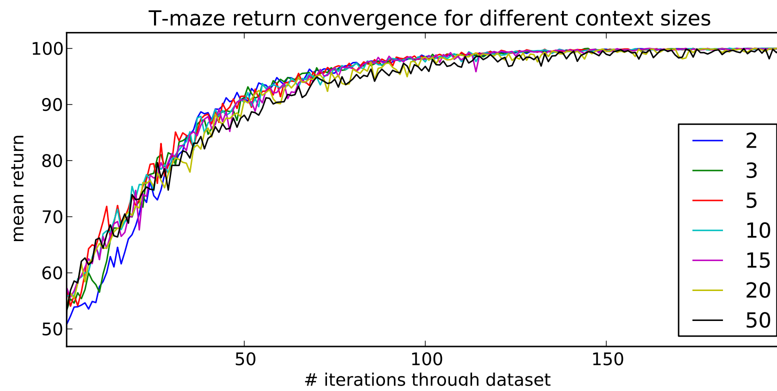


Figure 38: T-Maze convergence plots for varying numbers of context states. The mean over 30 repeated trials is plotted. All experiments converge at the same rate. Runs with large numbers of states exhibit erratic convergence curves towards the end. This phenomenon becomes more apparent when comparing the variances for these experiments (Figure 39).

parameter in the number of hidden nodes (amongst many other tunable parameters), Context Learning seems to express a meta-parameter in the number of context states. The minimum number of states can be gathered for simple problems like the T-Maze task, but for complex learning problems it is not clear what the number of context states should ideally be set to. The only proven boundary is that all regular languages (being a subclass of problems solvable with context learning) require only a finite number of states (Straubing, 1994).

8.3.2 Digits

In this experiment we classify 10 sequences of length 15 of binary values. The sequences are very well known and most people can instantly recognise and correctly classify each of them, when presented in a 3×5 grid, despite the fact that most of the sequences are very similar to each other and sometimes only differ by a single element. Figure 40 on page 123 shows the sequences both in grid and linear form. The linear form is a row-first linearization of the grid version. This experiment differs from the T-Maze problem in that it is not a binary but 10-class classification task.

Experiment Setup

The ten binary sequences (illustrated in Figure 40 bottom) are each sequentially presented to the context learning agent repeatedly. The val-

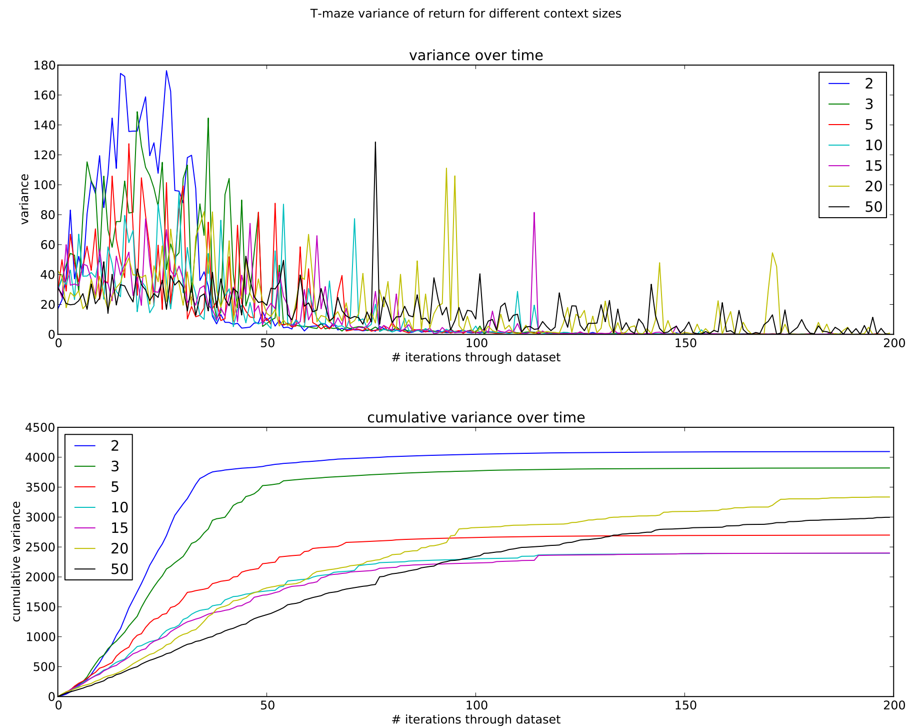


Figure 39: T-Maze convergence plots for varying numbers of context states. The variance (top graph) and cumulative variance (bottom graph) over 30 repeated trials is plotted. Trials with lower numbers of states start off with higher variance but converge faster and reach a plateau (close to zero change in variance), while more states lead to lower variance initially but do not reach the plateau as quickly.

idation set was identical to the training set in this case, consisting of all 10 episodes. We did not test for generalisation in this experiment because of the limited number of sequences. Instead, we look at convergence rate with different numbers of context states. As with the previous experiment, we use simple logistic regression classifiers for each context state. As this is again a sequence classification task (albeit with 10 instead of 2 classes), the importance vector for each sequence is 0.0 everywhere except for the last element, which is 1.0. Unless stated otherwise, the parameters for the Q-Learning agent are as follows: $\alpha = 0.1, \lambda = 0.9, \gamma = 0.9$. For exploration, discrete state-dependent exploration was used with $\epsilon_0 = 0.5$ decaying exponentially to $\epsilon_T = 10^{-4}$.

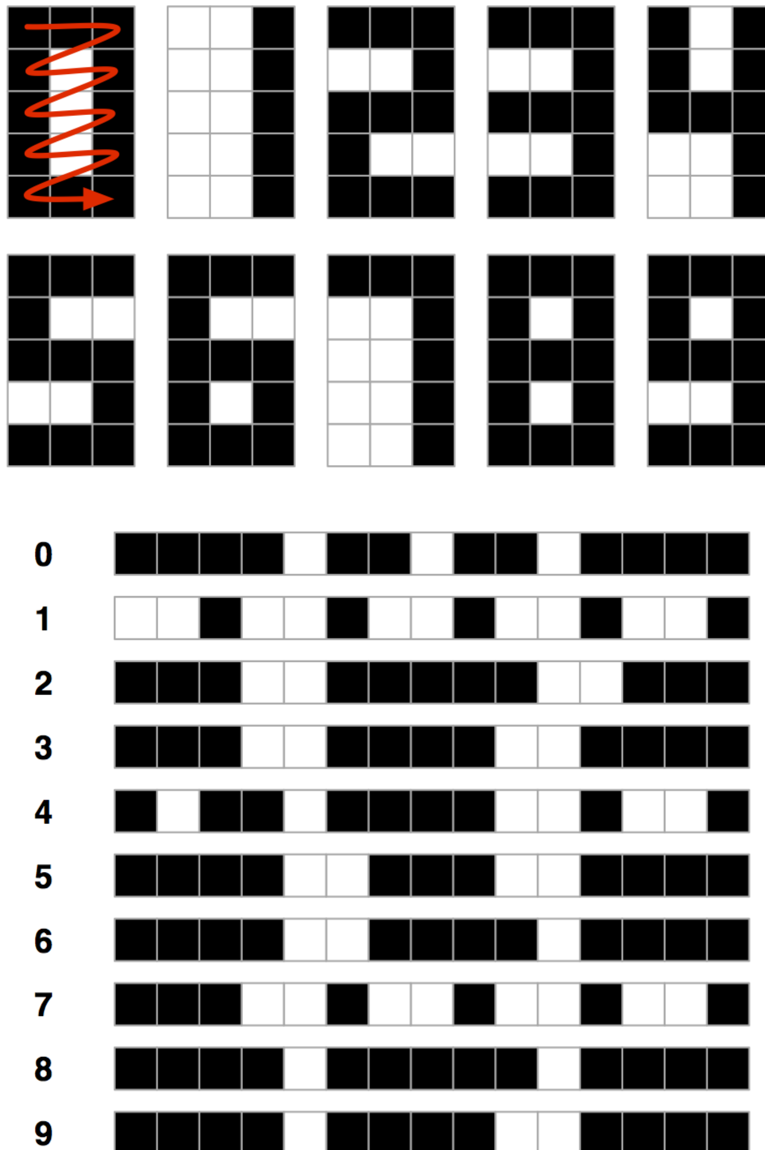


Figure 40: Digit sequences in 3×5 grid form above, and linear (row-major) below. The grid representation of digits is well-known for their use in alarm clocks and stop-watches with LCD displays. Despite their universal recognition, the sequences in linear form are very similar to each other, especially the 0, 2, 3, 5, 6, 8, 9 group of digits and the 1, 4, 7 group of digits.

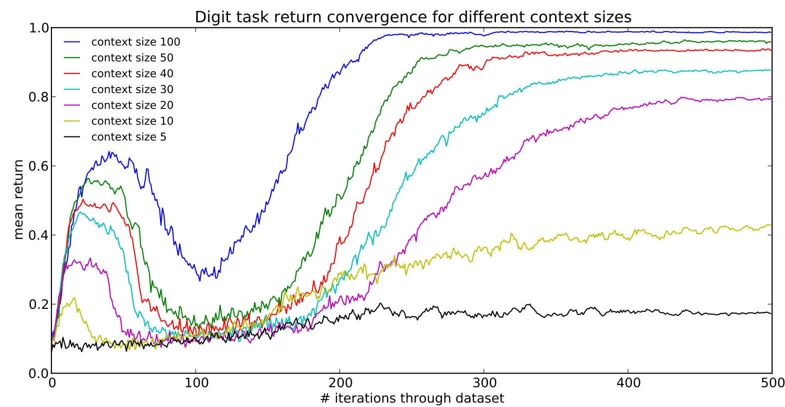


Figure 41: Convergence graphs for different context sizes in the Digit experiment (classifier learning rate was fixed at $\alpha = 0.1$). The larger the number of context states, the quicker the agent reaches convergence and with a higher end result. An agent with less than 20 context states was only able to marginally improve in performance. Since all sequences end with the same value (black pixel), it is impossible to solve the task with less than 10 context states. An interesting artefact of the experiment is the initial performance bump at around 50 iterations. The bump is higher the more context states are available. It can be explained as a premature convergence of the classifiers without corresponding improvement in the agent.

Results

The Digits experiment is more complex than the T-Maze task for several reasons: it is a multi-class problem with 10 different classes, rather than a binary decision problem, and the sequences are very similar to each other and do not have an obvious signal, which the T-Maze had. While the sequences of the T-Maze task were much longer in some cases (up to a 1000 steps), this actually does not impact the complexity of the problem for context learning, because context learning removes the temporal dependencies and solves the problem *orthogonal* to the sequence learning approaches, delivering a fixed point solution that works for any length of sequences.

Each of the digit sequences ends in a black pixel, therefore the classifier can only use information drawn from the context to decide on the class. This means that a minimum of 10 context states are required to solve the problem and we must therefore have 10 unique classifiers to perfectly solve the problem. While 10 states is the minimum, it is not clear if 10 are enough to solve the problem and we may even need more than 10 states.

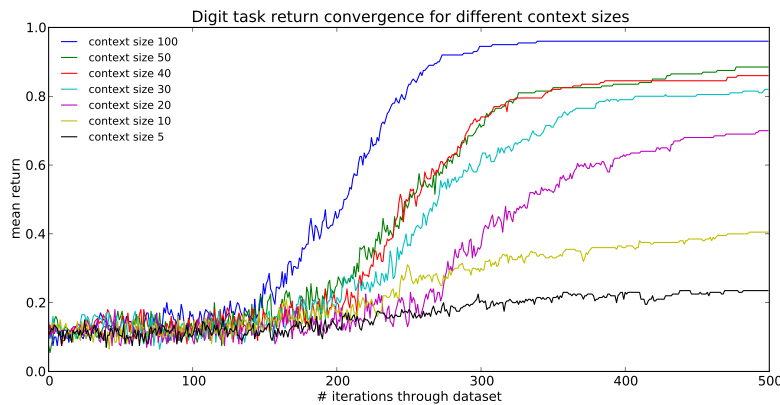


Figure 42: The same experiment as displayed in Figure 41 but with a lower learning rate of $\alpha = 0.001$ for all classifiers. The early bump in the convergence plots disappears, but overall performance suffers slightly.

This reasoning is confirmed in the plot in Figure 41, showing that 5 (black plot) or 10 (yellow plot) context states are not enough to get close to an optimal solution. Even though the 10-state graph is not fully saturated yet after 500 iterations, it is unlikely that the results would exceed a mean return of 0.45, which is a 45% rate on average of correctly recognising the digit sequence. Going up to 20 context states, however, there is a big increase in performance, yielding close to 80% correctness. The more context states we add, the better the results get, yet the difference in performance gain quickly decreases, to a point where doubling the number of states from 50 to 100 does only increase the final result by about 2% (compare blue and green curve at 500 iterations in Figure 41).

An unusual and interesting artefact of this experiment manifests as an initial bump in the convergence plots at around 50 iterations, being more prominent the more context states were used. A possible explanation for this bump is that there are two processes co-converging simultaneously: the reinforcement learning agent and the classifier. Initially, the transitions between states is random, and so is the final state for each sequence. However, because the exploration strategy is state-dependent and therefore deterministic with respect to a fixed sequence of states, the same sequences end up in the same final states. With more context states available, the chance of scattering all ten digits into disjunct final states is high, and the classifiers at these final states can adapt quickly to a sub-optimal solution of classifying the digits correctly. Once the agent becomes better in learning the state transitions, the final states change and other classifiers have to learn the correct target. This is where performance drops again, at around 100 iterations.

In short, because of the deterministic nature of the sequences, the classifiers adopt too quickly and are ahead of the agent. To verify this assumption, we ran the same experiment again but with a much smaller learning rate for the classifiers. Reducing the learning rate from $\alpha = 0.1$ for the original experiment series in Figure 41 to $\alpha = 0.001$, the suboptimal bump disappears (Figure 42). While the plots look more conventional with such a low learning rate, the reduction also has an effect on the final result, reducing the overall recognition rate of all runs by a few percentage points. Since the higher learning rate had no adverse effects other than an unusual convergence, it is therefore recommended to keep the learning rate high for optimal results.

Finally, the Digits task, while more complex than T-Maze, still is at a combinatorical level that allows introspection into the internals to understand what the agent actually learns. After the learning process has completed, we can switch off exploration and feed each sequence of digits into the deterministic agent and observe its behaviour. In particular, we are interested in the action choices, i.e., what state transition the agent proposes for different inputs. Mapping these action traces in a different colour for each of the digits, we get an interesting transition graph in Figure 45 that shows the agent's reasoning about different sequences. Not surprisingly, with a limited number of available context states, the agent was forced to re-use some of the nodes and paths through the graph efficiently. Figure 45 shows an experiment with 50 available context states, of which the agent only used 36. All sequences but the one for digit 1 start at the black square, and digit one (having a white pixel at its top left corner) starts at the white square. Gray squares stand for intermediate states, and the doubly lined circles represent final states, labelled with their matching digit class (learned by the classifier). Similar sub-sequences are re-used by several of the digits, for example sequence 2 (light green) and 7 (brown) share their initial 6 nodes, but the remaining sequence of 7 looks more like the sequence of 1 (*white, white, black, white, white, black*). As can be seen in the graph, the brown path then joins digit 1 (red), exiting the final node for 4, until it reaches its end in final node 7, whereas the red path continues for another *white, white, black* sequence until it reaches its own final node. It is interesting to see how resourceful nodes are re-used almost as if by design.

8.3.3 6-Bit Parity

This experiment compares Context Learning with Probabilistic Incremental Program Evolution (PIPE) (Salustowicz and Schmidhuber, 1997; Salustowicz, 2003). The publication evaluates PIPE in the 6-bit parity experiment, where the algorithm has to discover if a sample of length 6

bits has an odd or even number of 1 values. To be able to directly compare the PIPE against Context Learning, the experiment is repeated as described in the original publication. It should be noted, however, that simple experiments like 6-bit (or even n-bit) parity have a small search space and can actually be solved by simply guessing the parameters (e.g. the weights of an RNN) repeatedly until an adequate solution is found (Hochreiter and Schmidhuber, 1996; Schmidhuber et al., 2001).

Experiment Setup

Following the experiment setup described in Salustowicz and Schmidhuber (1997), the training and validation dataset was identical, consisting of 64 samples, one for each binary 6-bit representation. The target for each sample is 1 if the number of 1s in the sample bit string was odd, otherwise the target is 0. Some example samples and their target values:

$$\begin{aligned} &010110 \rightarrow 1 \\ &000110 \rightarrow 0 \\ &010000 \rightarrow 1 \\ &000000 \rightarrow 0 \\ &111111 \rightarrow 0 \end{aligned} \tag{8.5}$$

The fitness of a program is calculated by evaluating the program on all 64 samples. The worst (best) fitness is 0 (64) for getting all targets wrong (correct). Likewise, we will evaluate the agent performance by counting the correct answers when presenting all 64 samples.

The remaining setup for Context Learning is straight forward and similar to the T-Maze task. The agent walks through the bit string, building up context as it sees the individual bit values. Each bit string ends with a terminal symbol, upon which the agent has to make a decision, which is a simple classification task of the context value into the classes 0 and 1. To classify correctly, the agent has to rely solely right context, as all terminal symbols are identical and no information based on the terminal can be gained. The problem can be solved with 2 context states, which is the number being used in this experiment. Larger numbers did not worsen the results as determined in preliminary experiments with up to 10 context states.

The experiment was repeated 100 times, each run was executed for 300 episodes. These episodes can be seen as the equivalent to PIPE's program evaluations (PE), where the maximum was set to 500,000. The

Table 5: Results on the 6-bit Parity Experiment

Algorithm	Perfect Solution Found	# Evaluations min	# Evaluations median	# Evaluations max
Context Learning	**93%**	**105**	**163**	**289**
PIPE	70%	9,432	52,476	482,545
GP	60%	64,000	120,000	396,000

computational complexity of the two algorithms are not being compared in this experiment, instead we assume that a PIPE program evaluation is equally expensive to a Context Learning episode as both require to inspect all 64 samples once. One PIPE learning episode requires several computational steps after a program evaluation, namely "Learning from Population", "Mutation of the Prototype Tree" and "Prototype Tree Pruning". Context Learning has to evaluate all returns and update the Q-Table as well as the classifier parameters. Without going into very low-level implementation details, the assumption that one iteration for each method is equally computationally expensive should be fairly reasonable and not disadvantage either method.

All other Context Learning parameters were identical to the T-Maze task (see Section 8.3.1).

Results

Table 5 sums up the quantitative results for Genetic Programming (GP), PIPE, and Context Learning. The results for GP and PIPE are taken from Salustowicz and Schmidhuber (1997). A more detailed version of the algorithm and experiment description can be found in Rafal Salustowicz' Doctoral Thesis (Salustowicz, 2003).

Context Learning finds the perfect solution (classifying all 64 samples correctly) within the maximum allowed 300 iterations in 93 out of 100 runs. This number is 23% higher than PIPE's result, which was further allowed to execute many more program evaluations (500,000 instead of 300). This shows two things: Context Learning finds a perfect solution more frequently, and does so with far fewer computations and less data consumption (as both a Context Learning episode and a PIPE program evaluation have to look at each sample of the dataset once). The graph in Figure 43 shows the average convergence of the 100 repetitions of the Context Learning algorithm visually.

The n-bit parity problem is a great example for a problem that benefits from learning explicit context. It has discrete state values (only 0 and 1) and a perfect solution only requires two context states. It is not surprising, that Context Learning is ideally suited for this kind of problem

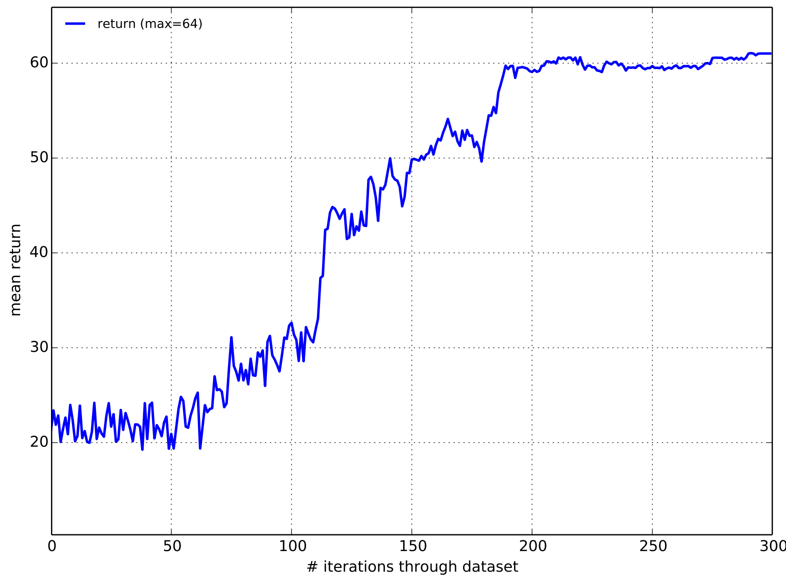


Figure 43: Convergence graph of Context Learning applied to the 6-bit parity problem. The graph shows average returns (maximum is 64) over 100 repetitions. Each repetition runs for 300 iterations (x-axis). 93 out of the 100 repetitions converge to a perfect solution with score 64, the remaining 7 are included in this graph and explain the overall result of just above 60.

and outperforms any kind of genetic programming approach, including PIPE, by far.

PIPE additionally uses an Elitist approach, protecting the fittest solution in the population from being destroyed by random exploration (permutation). Context Learning does not have this mechanism, and during the experiment it was often the case that a perfect solution was found early on but destroyed again due to exploration. Across all 100 repetitions, the average timestep at which a perfect solution was evaluated for the first time is after only 27.39 steps. It is a property of the algorithm that exploration is still fairly high early on (it is reduced exponentially over time) and that a good solution will temporarily be lost again. Therefore, having an elitist mechanism gives PIPE an unfair advantage in this comparison. Despite that, Context Learning still outperforms PIPE by several orders of magnitude, finding a perfect solution more than 20% more frequently, and using a fraction of the evaluations (0.31% in the median) that PIPE needs.

It should be pointed out though that Context Learning is currently limited to problems with discrete states that are classification tasks, whereas

PIPE can be applied to a more broad range of problems including regression.

8.4 DISCUSSION

This chapter introduced the concept of Context Learning as a novel way to classify sequences. At each time step through the sequence, a Reinforcement Learning agent guesses a context based on previous context and current input. A classifier then processes the context-enhanced input and feeds the success/failure back to the reinforcement agent as reward signal.

We've demonstrated its capabilities on three different tasks, the T-Maze benchmark, a digital digit recognition problem and 6-bit parity. The results of T-Maze showed that context learning — unlike other popular solutions like Recurrent Neural Networks — solves the problem independent of time by creating a fix-point solution that works for any corridor length. Algorithms using recurrent gradient-based policies are limited by the number of steps they can remember, due to the vanishing gradient problem. Context Learning does not suffer from the same limitation. It also required a vastly reduced number of iterations to learn the problem, two orders of magnitudes less, than the reference algorithms.

The second task was a binary sequence multi-class task, classifying digital clock digit sequences (3×5 matrix) into their corresponding digits 0–9. This problem let us explore the algorithm's behaviour on a more complex task, in particular the effect of different numbers of context states, and the close relationship between Context Learning and Finite State Machines.

Finally, the results on the 6-bit parity benchmark show the superiority of Context Learning over the related algorithm PIPE, but is currently limited to discrete problems, whereas PIPE can handle regression task as well.

For the remainder of this section, we discuss some aspects of Context Learning, implementation choices, their reasoning and possible alternatives.

8.4.1 Context Representation

In the introduction (Section 8.1.1) context was described as unobserved information that is used for classifying an object. What that means is

that the context does not tell us more about the object itself. Rather, it helps distinguish two objects, that appear the same (due to missing information) but are actually different. Going back to the T-Maze task, the agent only sees one of four symbols at each time step: c, n, s, x. In particular, every sequence ends with an x, yet the decision needs to be different depending on the history. The context needs to enhance the input of x and make it distinguishable for the two different cases. This can be achieved in different ways.

For this particular implementation of Context Learning, we used a simple state representation of context: the context is always in one of n disjunct states. The context by itself is meaningless (it is equivalent to a single integer number). But it helps distinguish inputs. For example, in context state 1, x can be perceived as x_1, and in context state 2, x is perceived as x_2. The numbers could be reversed, yet the outcome would be the same: The two x are now distinct from each other and the classifier can learn to act differently when confronted with either of them.

We've decided to use separate classifiers for each context state, that learn independently from the others. This gives the greatest flexibility for each of the cases, but it also means that the classifiers are not able to share any joint knowledge. Context could be represented in a different way, for example by concatenating input and context value, and feeding the combined input to a single classifier. This would have the added advantage that the classifier can contain "common knowledge", that is valid for all sequences, in addition to context-dependent knowledge.

An implementation of this idea shows that the context representation doe not matter much. For the results below (Figure 44), a single classifier (green line) received the input together with a one-of-n coded bit vector representing the current context state. It converges after ca. 300 episodes, similar to the original context representation with multiple classifiers (blue line). The early convergence bump is less strongly expressed, probably due to the fact that all solutions share a single classifier and early solutions are not as easy to learn as before, where each classifier was trained separately.

8.4.2 Sub-Sequence Classification

One of the characteristics of context learning is that each trace through context state space is unique (except for collisions; we will discuss this below) to the underlying sequence. Each prefix of a sequence influences the next context state, and subsequently the final context state. In the previous experiments, we used this property to classify full sequences. Even the slightest difference within a sequence would result in a differ-

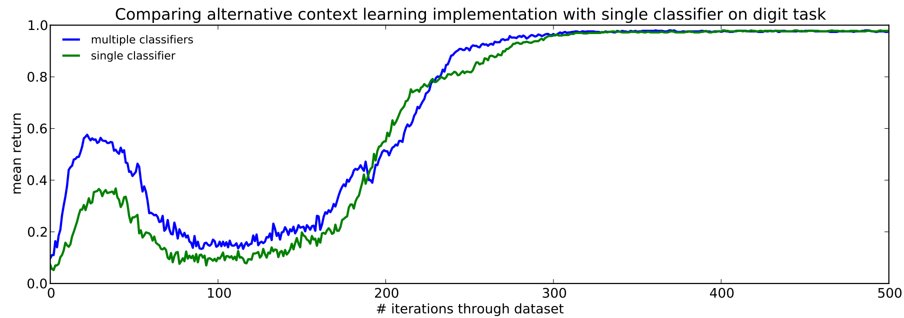


Figure 44: Two different implementations of context. The blue curve shows results for multiple classifiers, one per context state. Each classifier receives the same input, but can be trained with different targets depending on the context. The green line shows the implementation with a single classifier, but context-enhanced input. The context is encoded as a one-of-n vector (1 bit active) and concatenated to the input before passing it on to the classifier. Both implementations converge with similar overall rate.

ent final context state. This trait is useful for some applications, but the cases where we have perfectly measured, noise-free sequences from beginning to end are rare. Furthermore, it is often the case that not the full sequence, but rather short sub-sequences are indicative of a class. Many sequential problems are of this kind: letters form words in a sentence, words form sentences in a text, notes form melodies in music, nucleotides form genes in our DNA genome. In all these cases, rather short (compared to the full sequence) sub-sequences are the building blocks of the whole sequence. Due to the combinatorial variety, two sequences are almost never exactly identical to one another. Classification of the full sequence is therefore not feasible.

One approach to tackle this problem (one that works reasonably well for text) is the *bag of words* approach. The English language consists of estimated over 600,000 words according to the Oxford English dictionary (Simpson and Weiner, 1989). One can build a sparse vector of all the words, where each element represents the number of occurrences of the particular word in a text. Usually, stop words (like "the", "a", "with", etc.) are removed first and the remaining words are *stemmed*, which means their endings are removed and only the word stem is used. Common stemmers are available and implemented in many programming languages, for example the Porter stemmer (Van Rijsbergen et al., 1980). The remaining very sparse word vector can then be used for classification of texts, and simple classifiers like Naive Bayes work reasonably well. This approach, however, removes any positional information about the words. They are taken out of their sequential order and are replaced by simple counts of their frequency in the text.

While natural languages have enough words to make this strategy work, this would not be possible for music ("bag of notes") or DNA ("bag of nucleotides"), because the alphabet is much smaller. Arguably, the order of words in a sentence, while useful, does not seem to matter too much if a text is to be classified by its topic. But for other language tasks, like translation, bag-of-words does not help. One key difficulty in automated text translation is the disambiguation of words. Compare these two sentences.

> I'm looking forward to a warm *spring*.
>
> My bed could use a new *spring*.

We understand right away, that one of the springs refers to a season, and the other refers to a metal spiral. Machine text translation algorithms often try to solve this problem with n-grams, considering translations of tuples of n words instead of single words. But what about sentences that have the key operative words far apart from each other? An algorithm that would build a unique context while walking through the sentence might help to disambiguate the meaning of the word *spring*.

A "bag-of-notes" approach would not work for music recognition either. The modern diatonic scale consists of only 7 distinct notes. With such a small alphabet, the order is much more relevant than it was for text. Ordered sequences of notes are the essence of music and without the order it would be very hard if at all possible to distinguish characteristic songs. The same applies for the 4 nucleotides in DNA.

As described above, Context Learning assigns a specific context to each sequence. This is also true for any prefix of a sequence. At each point in time t within a sequence, we can view the current context state c_t as a *hash* of the sub-sequence from start to t (Attenberg et al., 2009). One single walk through a sequence of length T would give us hashes of all prefixes $\{(x_1, \ldots, x_t)\}_{t=1}^{T}$. To get a hash of every sub-sequence in a sequence of length n, we would have to walk the sequence n times, each time starting at a different element. This would make for a quadratic algorithm complexity. Limiting the hashes to sub-sequences of maximum length k, the algorithm requires $k \cdot n$ steps, making it linear in big-$\mathcal{O}$ notation. Similar to bag-of-words, this "bag-of-subsequences" approach would create a frequency count of sub-sequences in the full sequence and would allow to classify noisy sequence patterns.

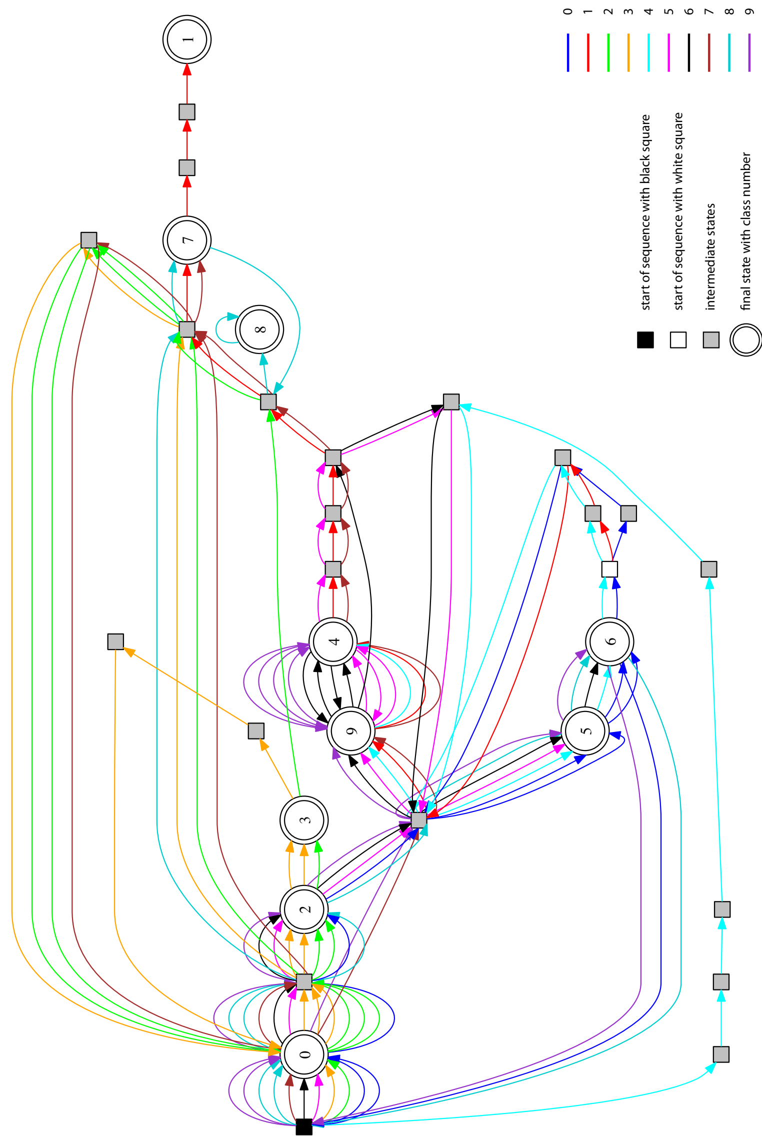


Figure 45: Learned decision graph of a fully trained context learning agent. Entry points are the black (all digits except 1) and white square (digit 1). The grey squares represent intermediate states, double circles are final states, labelled with the digit class. The agent had 50 context states available, of which it used 36. All final states are disjunct, making it very easy for the classifier to decide on the class. Sub-paths in the graph are efficiently re-used in some cases, for example for digit 3 (orange) the path sequence of 0, “grey square”, 2, 3 is traversed several times.

Part IV

Conclusion

9 DISCUSSION

In Chapter 6, we introduced a novel form of exploration for Reinforcement Learning (RL) as an alternative to the commonly used random exploration. While its main focus was the application to Policy Gradient Reinforcement Learning, an alternative version for Value-Based RL was presented as well.

State-Dependent Exploration (SDE) creates policy variations rather than adding additive randomness to each individual action. This approach inserts considerably less variance into each episode leading to faster convergence. SDE is also much more robust to environmental noise and exhibits advantages especially during longer episodes. In problems involving many tunable parameters it not only converges considerably faster than conventional exploration, but can also overcome local minima where random exploration can get stuck. In a robotics simulation task, SDE could clearly outperform state-of-the-art exploration techniques and delivered a stable, near-optimal result in almost every trial. Furthermore, SDE is simple and elegant, and easy to integrate into existing policy gradient implementations. The toy experiments serve to illustrate basic properties of SDE, while the physics-based ball catching simulation gives a first hint of SDE's performance in real-world applications.

While State-Dependent Exploration is not directly related to the following chapters, it turns out to be a helpful, if not necessary pre-requisite for subsequent algorithms which require an efficient deterministic exploration strategy.

The ongoing inspiration for Chapters 7 and 8 was drawn from the curiosity and fascination about the mechanics of human information processing and understanding. This research does in no way attempt to explain any aspects of the human mind, it is purely bio-inspired. Chapter 7 and 8 each contain one building block of a process, that on a high abstraction level may resemble that of human perception. Below we will map out how these ideas relate to the contents of the two Chapters.

The human mind is capable of making sense of its inherently continuous, multi-dimensional and sequential environment. And while our sensory organs can process large amounts of raw information in parallel, there seems to be an information bottleneck in our conscious processing. This bottleneck becomes evident in many ways, be it through the subjective

feeling of current awareness that seems to separate past from future events, or even the simple fact that we constantly trick our brains into perceiving a continuous stream of images (a movie) by presenting pictures with a frequency of 25 frames per second or higher.

To overcome the spatial information overload in the visual system, for instance, evolution developed an attentive process that focuses on small portions of our visual field, retrieving bits of information sequentially, despite the illusion of processing visual information as a whole. According to Kandel et al. (2000), the foveal region in the human eye, responsible for sharp vision and colour sensation, covers less than 1% of the retinal surface yet contains approximately 50% of optical nerves. It is only that small area that has a high enough resolution to recognise letters of text. Our eyes constantly execute saccades of rapid eye movement, processing word by word, or image by image, in sequence.

One of the research questions proposed in the Introduction in Chapter 1 was:

> *How can Machine Learning methods overcome the data processing bottleneck, focusing on relevant data and ignoring noise and useless information? How can we apply the concepts of human attention and selective cognition to learning algorithms?*

Chapter 7 centres around the attempts of integrating such a selective attention mechanism into general supervised learning tasks like classification to overcome potential bottlenecks in a data processing pipeline and to act as a filter for incoming information. We found that this can be achieved in the context of the Reinforcement Learning framework where an agent optimises its behaviour in a defined environment over time based on a sparse and scalar reward signal. While Reinforcement Learning is commonly used for control tasks in robotics, scheduling problems or game play, here we applied it to supervised learning tasks. We created a hybrid system that can simultaneously learn an attentive control mechanism to steer focus towards informative portions of data while learning the mapping function from input to class. Instead of relying on commonly used static feature selection as a pre-processing step, this approach can be seen as sequential online feature selection, that chooses features on the fly, depending on the current state of the supervised task it interacts with.

In more detail, we have derived classification as a (Partially Observable) Markov Decision Process and thus made it accessible to RL methods. The application we focused on was minimisation of data consumption, by training an RL agent to pick features first that lead to quick classification. We presented results for different classifiers (both static and sequential) on vision and medical tasks. Our approach reduces the num-

ber of necessary features to access to a fraction of the full input, down to 12% with RNN classifiers. We also demonstrated that this approach is able to deal with weighted feature costs, a property that is useful in plenty of real-world applications. Furthermore, a new action selection method was introduced that draws actions without replacement. It should prove useful in other ordering tasks as well, such as scheduling problems.

The resulting system is capable of processing spatial information sequentially, filtering out irrelevant, noisy, misleading or redundant information and focusing on the portions of data that help with the supervised learning task.

Human processing of sequential information seems to have no measurable limitation to the length or amount of data, nor the time period between connected pieces of information. Once observed, relevant data can be recalled and used for understanding or decision making minutes, days or sometimes years later.

To give an example, attentive readers who made it through the full length of this thesis may remember[1] the leading quote about *recursion* from the Introduction in Chapter 1. At the time of reading that specific sentence for the first time, it was not obvious that a later part of this book would refer back to it. Yet the memory of reading this quote is now readily available and helps comprehend this self-referential thought experiment.

This ability indicates that information is not processed in a fixed time window (keyword: n-grams), but rather that current information is evaluated in a context that encapsulates the history as a whole, giving access to any memory if your attention mechanism deemed it relevant enough to be remembered (which may or may not have been the case for you here).

The second research question addressed this specific point:

> *How can we learn an explicit representation of context independent of sequence length? Are there any alternatives to sequence learning that do not suffer from memory limitations?*

Chapter 8 takes after this idea and addresses the problem of sequential information processing in supervised problem domains. The proposed approach builds an explicit contextual state while traversing through the sequence. Unlike many alternative methods (like RNNs), Context Learning does not attempt to learn this context in supervised fashion, but rather treats it as opaque black box, aimed to distinguish current

1 Those who don't remember reading it (or remember *not* reading it) may cheat at this point and look it up on page 3.

ambiguous inputs. Reinforcement Learning again offers a convenient framework to describe the task as a reward-driven problem, exploring contextual states and improving the choices through feedback from the supervised component.

Context Learning can be regarded as the continuation of the work presented in Chapter 7, taking sequenced information as input and processing it further, to a point where it can be dealt with by regular, non-sequential supervised learning methods.

Finally, all three Methodology chapters (Chapters 6–8) share in common a tight integration of Reinforcement Learning and Supervised Learning components. In Chapter 6 we use a function approximator to represent the state-dependent exploration function for direct RL. Chapter 7 uses a supervised classifier as source of reward for a RL controller to learn to select the next feature. Finally, in Chapter 8 a RL agent builds a context state to augment the input to a supervised learning algorithm to distinguish states without their temporal dependencies.

This brings us to the third research question, which is not confined to a single chapter but is threaded throughout all of Part III:

> *Can state-of-the-art Supervised Learning algorithms be improved by infusing them with Reinforcement Learning? And will the resulting hybrid algorithms perform qualitatively and/or quantitatively better on established benchmarks than the standard algorithms alone?*

To answer this final question, we have compared each of the algorithms with existing state-of-the-art methods and showed how such hybrid alternatives outperform their purebred counterparts. It was further shown that Reinforcement Learning can successfully be applied in predominantly data-driven environments outside of its usual fields of application like robotics or game play. Removing slow real-life interaction (i.e. the feedback loop of physical robot control) allows the state-action-reward cycle to be executed purely on computer processors, an ideal habitat for such data-hungry algorithms. Another interesting and reoccurring theme was the iterative co-convergence between RL and Supervised methods, conceptually similar to Expectation-Maximisation algorithms (Dempster et al., 1977), here, however, alternating between supervised training and policy improvement. Finally, we hope that this work has laid the foundation for future hybrid Reinforcement/Supervised learning implementations and has helped to mature Reinforcement Learning into a true alternative for solving problems that were traditionally approached only with supervised learning.

10 FUTURE WORK

This thesis presented three major contributions: State-Dependent Exploration (Chapter 6), Sequential Feature Selection (Section 7) and Context Learning (Chapter 8). While they can be regarded as parts of a greater system, they also stand as independent contributions, with their own unresolved questions and open problems. As such, for this final chapter, we will give an outlook on future work for each part separately, before we close with remarks on the combined system in Section 10.4.

10.1 STATE-DEPENDENT EXPLORATION

10.1.1 Extending Exploration Functions

In its current implementation, SDE only considers addition as the operator to combine the controller with the exploration function, see Eqn. (6.7). Future research could investigate alternative operations, like multiplication and function chaining, leading to actions determined by:

$$a = f(s,\Theta)\, \epsilon(s,\hat{\Theta}) \tag{10.1}$$

$$a = \epsilon \circ f = \epsilon(f(s,\Theta),\, \hat{\Theta}) \tag{10.2}$$

Eqn. (10.1) describes a strategy that scales the original actions based on some exploration parameters $\hat{\Theta}$ for each state s. This could lead to interesting possibilities, like disabling exploration in certain areas of state space, having $\epsilon(s,\hat{\Theta})$ return constant 1 for certain values of s, and strengthening and weakening actions with a simpler family of exploration functions than would otherwise be required.

Eqn. (10.2) is probably the most flexible approach, chaining exploration and controller together. The exploration function ϵ would take the deterministic output of the controller as input. Depending on the choice of ϵ, any result would be possible, and it's hard to say what properties and benefits (if any) this approach has, but it is certainly worth investigating in the future.

10.1.2 Learning Exploration Variances

Another shortcoming with any non-linear exploration function is that the variance to control the amount of exploration may no longer be derivable from the log likelihood. The nice property of adaptive exploration that holds for linear exploration functions is then not available, and manual hand-tuning was suggested as the alternative. But linear exploration functions are limited in how they can modify the agent's behaviour. One interesting line of future research would be to see if exploration variances can be learned, either in supervised fashion (although it is not immediately clear what the target would be) or with Reinforcement Learning. For the latter, it may be possible to come up with a modified reward function, that takes exploration variance into account as a secondary objective.

10.2 SEQUENTIAL ONLINE FEATURE SELECTION

10.2.1 Stopping Criterion Alternatives

One of the main difficulties for Sequential Feature Selection was to find a suitable stopping mechanism. SOFS has interesting properties by itself, without knowing by itself when to classify, but it felt like an incomplete method without this ability. Especially when comparing to existing methods, SOFS does not fit into the usual *pre-process and classify* pattern. One way out of this dilemma was to re-formulate SOFS as a MDP and combining classifier and feature selector into one inseparable algorithm. This lead to a method that was able to selectively choose features and classify the input pattern, but another initial design choice, that of modularity got lost on the way. Future work on SOFS could explore the possibilities of finding another way to learn when to stop accumulating features and to classify the existing pattern, despite the various unsuccessful attempts presented in Section 7.4. A desirable outcome would keep the attention mechanism separate from the classifier while still intrinsically knowing when enough information is available to predict the class.

10.2.2 Classifier State Representation

In its current form SOFS (without integrated classification) uses the belief state of the classifier as input to the attention learning agent. This decision, while motivated (see Section 7.3.5), was a somewhat arbitrary

choice that happened to work well. The state is limited to what the classifier believes to see, but does not reveal any information about the current input. By giving the RL agent access to current input, it could potentially make use of that information directly, rather than channelling it through the classifier into the belief state. However, there is a fine line here between revealing too little and too much information. In the case of Integrated Classification (SOFS+IC), the RL state must contain all sensory input as the agent is both in charge of attention and classification. This increases the input space for the RL component tremendously, as pointed out in Section 7.4.2, making the learning problem much more complex. Here, future work should focus on finding alternative classifier representations and finding a good middle ground between the two extremes.

10.2.3 Suitability for Regression Tasks

One aspect that both Chapters 7 and 8 have in common is the focus on Classification tasks as supervised learning component. For Context Learning, the choice of supervised task does not make any difference, as the only feedback to the RL agent is how well the task was solved across all samples. Adapting Context Learning to regression tasks only requires a simple drop-in replacement of the supervised learning method. For Sequential Feature Selection, however, the procedure is more involved. The reason for this is that the SOFS agent requires an internal state of the supervised learning component as its input. In its original form, SOFS uses the vector of beliefs over all class memberships as input. Such a vector is not available in regression. One could think of an adapted version using expected mean and variance for the predicted value as input to the agent, but such changes need to be thoroughly assessed theoretically and evaluated through experiments.

10.3 CONTEXT LEARNING

Chapter 8 only touched the surface of the idea of Context Learning. There are many still unexplored dimensions that are not covered in this thesis, and many more tests and experiments are necessary to prove its usefulness and validity. We will discuss some of these aspects in the sections below.

10.3.1 Real-World Applications

The experiments presented in this chapter can only be considered a proof of concept. They demonstrated the basic properties of Context Learning and let us see some interesting aspects in regard to changing context size. We also showed their superiority over temporal methods like RNNs for the T-Maze benchmark. The next step now is to apply Context Learning to more complex tasks. A few well-suited fields were mentioned before: word sense disambiguation and text translation, song recognition and DNA sequence classification are promising candidates for context learning, as they all require contextual information or work on small alphabets, where a sole frequency based bag-of-words approach is not an option. Sub-sequence context learning as discussed in 8.4.2 will likely play an important part in the application of this method to real-world problems.

10.3.2 Continuous Outputs

Chapter 8 has only looked at classifiers as the supervised learning component. But Context Learning is not limited to classification. Sequence prediction and generation are more closely related to regression methods, where based on the history of the sequence and the current input, one tries to predict the next element in the input sequence, or an element of a target sequence. In both cases, building context while walking through the sequence would remain identical, with a regression method taking the place of the classification method. The key difference is that the supervised target becomes a continuous value, whereas it was discrete for the classification version of Context Learning.

10.3.3 Continuous Inputs

Our examples in this chapter contained discrete input sequences. Many time series tasks have continuous real-valued and multi-dimensional inputs, however. This affects both the supervised and the reinforcement learning component. Replacing the discrete RL algorithm with a continuous alternative is possible, but as described in Section 5.3.2 continuous RL algorithms do not have the same convergence guarantees that discrete methods have. Direct RL methods (Section 5.4) may be a viable alternative here. In general, they are more data-hungry, but this is less of a problem in pure data-processing applications like these, where no physical hardware (e.g., a robot) is involved, and where data is available in abundance.

10.3.4 Continuous Context

The representation of context in its current implementation is discrete. Context is always one of a finite number of states. This assumption made the Reinforcement Learning part easier, as we could pick a method working on discrete actions. Continuous Context does have some appealing properties, and exploring its feasibility is certainly worthwhile. Continuous context could be defined on a similarity metric, leading to similar context for similar input states. Function approximators could make use of this property and learn a smooth distribution of actions. Knowledge transfer from unseen but similar past situations would speed up learning as well.

Section 5.3.2 describes methods for continuous action space and explains the difficulty of finding the action with highest expected reward. Yet some methods exist and would be suitable for this task. Direct Reinforcement learning algorithms again could be a good match here.

10.3.5 Beyond Deterministic Finite Automata

Section 8.2.1 derived Chomsky Type-3 regular languages and deterministic finite automata as a special case of Context Learning (as it was defined in this chapter). An interesting extension of this work could explore the possibility of climbing up the Chomsky hierarchy of formal languages further, for instance by implementing equivalent algorithms for Push-down Automata (Type-II context-free languages). An RL agent would at each point in the sequence decide, based on current input, to store information on a stack or to retrieve information from the stack, at which point it would be made available to the classifier as part of the input. This path could potentially lead to the ultimate goal of training Turing machines to construct the necessary context, which are known for their universal computational properties.

10.4 COMBINING SOFS WITH CONTEXT LEARNING

Lastly, while the here presented methods (mainly Chapters 7 and 8) are designed to work hand in hand with each other, the combined system of SOFS and Context Learning has not been implemented and tested yet. In theory, the overall system should be able to process high-dimensional information by turning it into a sequence of relevant data points, then processing those by building meaningful context throughout the sequence. An evaluation against methods like MDRNNs (Graves

et al., 2007) would be of interest, since they also sequentialize and classify multi-dimensional data. MDRNNs consume the entire sample multiple times, therefore SOFS would be of great benefit in terms of data consumption.

Another interesting aspect to investigate would be if a parallel execution of the two algorithms can also mitigate the need for an explicit stopping criterion (as discussed under Section 10.2.1), as the classification would happen simultaneously to the online feature selection, classifying whatever is currently available in the feature pipeline.

Overall, as it is often the case in scientific studies, this research has answered some questions but raised many more. We believe that the work presented here, in particular the youngest (and least refined) topic of Context Learning, still has significant undiscovered potential worth investigating further. It is our hope that the ideas presented in this thesis inspire other researchers to continue this quest to bring us closer to a much sought-after general, human-level Artificial Intelligence.

BIBLIOGRAPHY

P. Abbeel and A.Y. Ng. Exploration and Apprenticeship Learning in Reinforcement Learning. In *Proceedings of the 22nd international conference on Machine learning*, page 8. ACM, 2005.

P. Abbeel, A. Coates, M. Quigley, and A.Y. Ng. An Application of Reinforcement Learning to Aerobatic Helicopter Flight. *Advances in Neural Information Processing Systems: Proceedings of the 2006 Conference*, 2007.

D.A. Aberdeen. *Policy-Gradient Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Australian National University, 2003.

Shun-Ichi Amari. Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2):251–276, 1998.

A. Antos, R. Munos, and C. Szepesvari. Fitted Q-Iteration in Continuous Action-Space MDPs. *Advances in Neural Information Processing Systems*, 20:9–16, 2008.

J. Attenberg, A. Dasgupta, J. Langford, A. Smola, and K. Weinberger. Feature Hashing for Large Scale Multitask Learning. In *Proceedings of the International Conference of Machine Learning (ICML)*, 2009.

L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

L. Baird and A.W. Moore. Gradient Descent for General Reinforcement Learning. *Advances in Neural Information Processing Systems*, pages 968–974, 1999.

Leemon Baird and Harry A. Klopf. Reinforcement Learning with High-Dimensional, Continuous Actions. Technical report, Wright Laboratory, Wright-Patterson Air Force Base, OH 45433-7301, USA, 1993.

B. Bakker et al. Reinforcement Learning with Long Short-Term Memory. *Advances in Neural Information Processing Systems*, 14:1475–1482, 2002.

Dana H Ballard. Modular learning in neural networks. In *Proc. AAAI*, pages 279–284, 1987.

L.E. Baum, T. Petrie, G. Soules, and N. Weiss. A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains. *The Annals of Mathematical Statistics*, pages 164–171, 1970.

J. Baxter and P.L. Bartlett. Reinforcement Learning in POMDPs via Direct Gradient Ascent. *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 41–48, 2000.

J. Baxter and P.L. Bartlett. Infinite-Horizon Policy-Gradient Estimation. *Journal of Artificial Intelligence Research*, 15(4):319–350, 2001.

Loris Bazzani, Nando Freitas, Hugo Larochelle, Vittorio Murino, and Jo-Anne Ting. Learning Attentional Policies for Tracking and Recognition in Video with Deep Networks. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 937–944, June 2011. ISBN 978-1-4503-0619-5.

Richard Bellman. A Markovian Decision Process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

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

C.M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, USA, 1995.

C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Occam's razor. *Information processing letters*, 24(6):377–380, 1987.

J. Boyan and A.W. Moore. Generalization in Reinforcement Learning: Safely Approximating the Value Function. *Advances in Neural Information Processing Systems*, pages 369–376, 1995.

Arthur E Bryson. A gradient method for optimizing multi-stage allocation processes. In *Proc. Harvard Univ. Symposium on digital computers and their applications*, 1961.

Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI*, pages 183–188, 1992.

D. Cohn, L. Atlas, and R. Ladner. Improving Generalization with Active Learning. *Machine Learning*, 15(2):201–221, 1994.

Jerome T Connor, R Douglas Martin, and LE Atlas. Recurrent neural networks and robust time series prediction. *Neural Networks, IEEE Transactions on*, 5(2):240–254, 1994.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

M.P. Deisenroth, C.E. Rasmussen, and J. Peters. Gaussian Process Dynamic Programming. *Neurocomputing*, 72(7-9):1508–1524, 2009.

Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

S. E. Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.

S. E. Dreyfus. The computational solution of optimal control problems with time lag. *IEEE Transactions on Automatic Control*, 18(4):383–385, 1973.

G. Dulac-Arnold, L. Denoyer, and P. Gallinari. Text Classification: A Sequential Reading Approach. *Advances in Information Retrieval*, pages 411–423, 2011a.

G. Dulac-Arnold, L. Denoyer, P. Preux, and P. Gallinari. Datum-Wise Classification: A Sequential Approach to Sparsity. In *Proceedings of the European Conference of Machine Learning (ECML 2011)*, pages 375–390. Springer, 2011b.

J.L. Elman. Finding Structure in Time. *Cognitive science*, 14(2):179–211, 1990.

Ido Erev and Alvin E Roth. Predicting How People Play Games: Reinforcement Learning in Experimental Games with Unique, Mixed Strategy Equilibria. *American Economic Review*, pages 848–881, 1998.

D. Ernst, P. Geurts, and L. Wehenkel. Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6(1):503, 2005. ISSN 1532-4435.

D. Ernst, R. Maree, and L. Wehenkel. Reinforcement Learning with Raw Image Pixels as Input State. *Lecture Notes in Computer Science*, 4153: 446, 2006.

A. Frank and A. Asuncion. UCI Machine Learning Repository. University of California, Irvine, CA. http://archive.ics.uci.edu/ml/, Oct. 2011.

Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

C. Gaskett, D. Wettergreen, and A. Zelinsky. Q-Learning in Continuous State and Action Spaces. *Advanced Topics in Artificial Intelligence*, pages 417–428, 1999.

Romaric Gaudel and Michele Sebag. Feature Selection as a One-Player Game. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 359–366, 2010.

F. A. Gers, N. Schraudolph, and J. Schmidhuber. Learning Precise Timing with LSTM Recurrent Networks. *Journal of Machine Learning Research*, 3:115–143, 2002.

F. J. Gomez and J. Schmidhuber. Co-Evolving Recurrent Neurons Learn Deep Memory POMDPs. In *Proc. of the 2005 conference on genetic and evolutionary computation (GECCO), Washington, D. C.* ACM Press, New York, NY, USA, 2005.

A. Graves, S. Fernandez, and J. Schmidhuber. Multi-dimensional recurrent neural networks. In *Proceedings of the International Conference on Artificial Neural Networks ICANN*, 2007.

Alexander Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. PhD thesis, Technische Universität München, July 2008.

Vincent Graziano, Tobias Glasmachers, Tom Schaul, Leo Pape, Giuseppe Cuccu, Jürgen Leitner, and Jürgen Schmidhuber. Artificial curiosity for autonomous space exploration. *Acta Futura*, pages 1–16, 2011.

Mandy Grüttner, Frank Sehnke, Tom Schaul, and Jürgen Schmidhuber. Multi-dimensional deep memory atari-go players for parameter exploring policy gradients. In *Artificial Neural Networks–ICANN 2010*, pages 114–123. Springer, 2010.

I. Guyon and A. Elisseeff. An Introduction to Variable and Feature Selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

J. Hadamard. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*. Mémoires présentés par divers savants à l'Académie des sciences de l'Institut de France: Éxtrait. Imprimerie nationale, 1908.

D.I.R. Hafner. *Dateneffiziente Selbstlernende Neuronale Regler*. PhD thesis, Universität Osnabrück, 2009.

R. Hafner and M. Riedmiller. Reinforcement Learning in Feedback Control. *Machine learning*, 27(1):55–74, 2011.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

MT Harandi, MN Ahmadabadi, and BN Araabi. Face Recognition Using Reinforcement Learning. In *International Conference on Image Processing*, volume 4, 2004.

G. Hinton, S. Osindero, M. Welling, and Y.W. Teh. Unsupervised Discovery of Nonlinear Structure Using Contrastive Backpropagation. *Cognitive Science*, 30(4):725–731, 2006a.

Geoffrey Hinton and Ruslan Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, May 2006b. ISSN 0899-7667.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991. Advisor: J. Schmidhuber.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Sepp Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

Sepp Hochreiter and Jurgen Schmidhuber. Bridging long time lags by weight guessing and "long short-term memory". *Spatiotemporal models in biological and artificial systems*, 37:65–72, 1996.

RA Howard. *Dynamic Programming and Markov Processes*. Technology Press of Massachusetts Institute of Technology (Cambridge), 1960.

M. Hüsken and P. Stagge. Recurrent Neural Networks for Time Series Classification. *Neurocomputing*, 50:223–235, 2003.

MI Jordan. Attractor dynamics and parallelism in a connectionist sequential network. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 531–546. Erlbaum, Hillsdale NJ, 1986.

Eric R Kandel, James H Schwartz, Thomas M Jessell, et al. *Principles of Neural Science*, volume 4. McGraw-Hill New York, 2000.

Henry J Kelley. Gradient theory of optimal flight paths. *ARS Journal*, 30 (10):947–954, 1960.

A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11, 1965.

T. Kovacs and R. Egginton. On the Analysis and Design of Software for Reinforcement Learning, with a Survey of Existing Systems. *Machine learning*, pages 1–43, 2011.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

L.J. Lin. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8(3):293–321, 1992.

S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki, 1970.

F. Liu and J. Su. Reinforcement Learning-Based Feature Learning for Object Tracking. In *Proceedings of the 17th International Conference on Pattern Recognition*, volume 2, pages 748–751. IEEE, 2004.

H. Liu and H. Motoda. *Computational Methods of Feature Selection*. Chapman & Hall, 2008.

David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press New York, 2003.

F. Maes, L. Denoyer, and P. Gallinari. Sequence Labeling with Reinforcement Learning and Ranking Algorithms. *Machine Learning: ECML 2007*, pages 648–657, 2007.

K. Hornik Maxwell and H. White. Multilayer Feedforward Networks are Universal Approximators. *Neural networks*, 2(5):359–366, 1989.

R Andrew McCallum. Instance-based utile distinctions for reinforcement learning with hidden state. In *ICML*, pages 387–395, 1995.

M. Milano, J. Schmidhuber, and P. Koumoutsakos. Active Learning with Adaptive Grids. *Artificial Neural Networks—ICANN 2001*, pages 436–442, 2001.

J.R. Millán, D. Posenato, and E. Dedieu. Continuous-Action Q-Learning. *Machine Learning*, 49(2):247–265, 2002.

Shiwali Mohan and John E Laird. Learning to play mario. Technical Report Tech. Rep. CCA-TR-2009-03, Center for Cognitive Architecture, University of Michigan, 2009.

G.E. Monahan. A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms. *Management Science*, pages 1–16, 1982.

Luis Montesano and Manuel Lopes. Learning grasping affordances from local visual descriptors. In *Development and Learning, 2009. ICDL 2009. IEEE 8th International Conference on*, pages 1–6. IEEE, 2009.

J. Moody and M. Saffell. Learning to Trade via Direct Reinforcement. *IEEE Transactions on Neural Networks*, 12(4):875–889, 2001.

R. Munos. Policy Gradient in Continuous Time. *The Journal of Machine Learning Research*, 7:771–791, 2006.

Y. Nakamura, T. Mori, M. Sato, and S. Ishii. Reinforcement learning for a biped robot based on a CPG-actor-critic method. *Neural Networks*, 20 (6):723–735, 2007.

K. S. Narendra and M. A. L. Thathatchar. Learning automata – a survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323–334, 1974.

G. Neumann and J. Peters. Fitted Q-Iteration by Advantage Weighted Regression. *Advances in Neural Information Processing Systems*, 21:1177–1184, 2009.

G. Neumann, M. Pfeiffer, and H. Hauser. Batch Reinforcement Learning Methods for Point to Point movements. Technical report, Graz University of Technology, 2006.

E. Norouzi, M. Nili Ahmadabadi, and B. Nadjar Araabi. Attention Control with Reinforcement Learning for Face Recognition Under Partial Occlusion. *Machine Vision and Applications*, pages 1–12, 2010.

L. Paletta and A. Pinz. Active Object Recognition by View Integration and Reinforcement Learning. *Robotics and Autonomous Systems*, 31:71–86, 2000.

L. Paletta, G. Fritz, and C. Seifert. Q-Learning of Sequential Attention for Visual Object Recognition from Informative Local Descriptors. In *Proceedings of the 22nd International Conference on Machine Learning*, volume 22, page 649, 2005.

J. Pazis and M.G. Lagoudakis. Binary Action Search for Learning Continuous-Action Control Policies. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 793–800. ACM, 2009.

S. Perkins and J. Theiler. Online Feature Selection Using Grafting. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pages 592–599, 2003.

J. Peters and S. Schaal. Reinforcement Learning of Motor Skills with Policy Gradients. *Neural Networks, Special Issue*, 21(4):682–697, 2008a.

J. Peters and S. Schaal. Natural Actor-Critic. *Neurocomputing*, 71(7-9): 1180–1190, 2008b.

Jan Peters and Stefan Schaal. Policy Gradient Methods for Robotics. In *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.

Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural Actor-Critic. In *Proceedings of the Sixteenth European Conference on Machine Learning*, 2005.

DV Prokhorov, DC Wunsch, et al. Adaptive Critic Designs. *IEEE Transactions on Neural Networks*, 8(5):997–1007, 2002. ISSN 1045-9227.

Carl Edward Rasmussen and Christopher K.I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

M.L. Raymer, T.E. Doom, L.A. Kuhn, and W.F. Punch. Knowledge Discovery in Medical and Biological Datasets Using a Hybrid Bayes Classifier/Evolutionary Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 33(5):802–813, 2003.

M. Riedmiller. Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method. *Lecture Notes in Computer Science*, 3720:317, 2005.

Martin Riedmiller, Jan Peters, and Stefan Schaal. Evaluation of Policy Gradient Methods and Variants on the Cart-Pole Benchmark. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 2007.

Mark Ring. Learning sequential tasks by incrementally adding higher orders. *Advances in neural information processing systems*, pages 115–115, 1993.

Mark Ring. Child: A first step towards continual learning. *Machine Learning*, 28(1):77–104, 1997.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Thomas Rückstieß and Jürgen Schmidhuber. A Python Experiment Suite. *The Python Papers*, 6(1):2, 2011a.

Thomas Rückstieß and Jürgen Schmidhuber. Python Experiment Suite Implementation. *The Python Papers Source Codes*, 2:4, 2011b.

Thomas Rückstieß, Martin Felder, and Jürgen Schmidhuber. State-Dependent Exploration for Policy Gradient Methods. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases 2008, Part II, LNAI 5212*, pages 234–249, 2008a.

Thomas Rückstieß, Martin Felder, Frank Sehnke, and Jürgen Schmidhuber. Robot Learning with State-Dependent Exploration. In *1st International Workshop on Cognition for Technical Systems*, 2008b.

Thomas Rückstieß, Frank Sehnke, Tom Schaul, Daan Wierstra, Sun Yi, and Jürgen Schmidhuber. Exploring Parameter Space in Reinforcement Learning. *Paladyn Journal of Behavioral Robotics*, 1(1):14–24, 2010.

Thomas Rückstieß, Christian Osendorfer, and Patrick van der Smagt. Sequential Feature Selection for Classification. In *Proceedings of the Australasian Conference on Artificial Intelligence*, pages 132–141. Springer, 2011.

Thomas Rückstieß, Christian Osendorfer, and Patrick van der Smagt. Minimizing Data Consumption with Sequential Online Feature Selection. *International Journal of Machine Learning and Cybernetics*, 2012. ISSN 1868-8071. doi: 10.1007/s13042-012-0092-x.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

GA Rummery and M. Niranjan. Online Q-Learning Using Connectionist Systems, 1994.

M. Saar-Tsechansky and F. Provost. Handling Missing Values When Applying Classification Models. *Journal of Machine Learning Research*, 8(1625-1657):9, 2007.

Rafal Salustowicz. *Probabilistic Incremental Program Evolution*. PhD thesis, TU Berlin, 2003.

Rafal Salustowicz and Jürgen Schmidhuber. Probabilistic incremental program evolution: Stochastic search through program space. In *ECML*, pages 213–220, 1997.

A.L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. II – Recent Progress. *Annual Review in Automatic Programming*, 6:1–36, 1969. ISSN 0066-4138.

Nicholas Sapankevych and Ravi Sankar. Time series prediction using support vector machines: a survey. *Computational Intelligence Magazine, IEEE*, 4(2):24–38, 2009.

S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert. Learning Movement Primitives. In *International Symposium on Robotics Research*, 2004.

T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11: 743–746, 2010.

J. Schmidhuber. A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers. In *Proc. of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 222–227, 1991a.

J Schmidhuber. Curious model-building control systems. In *Neural Networks, 1991. 1991 IEEE International Joint Conference on*, pages 1458–1463. IEEE, 1991b.

J. Schmidhuber. A Fixed Size Storage O(n3) Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks. *Neural Computation*, 4(2):243–248, 1992.

J. Schmidhuber. Artificial Curiosity Based on Discovering Novel Algorithmic Predictability Through Coevolution. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.

J. Schmidhuber. Simple algorithmic principles of discovery, subjective beauty, selective attention, curiosity & creativity. Joint Invited Lecture for Algorithmic Learning Theroy (ALT 2007) and Discovery Science (DS 2007), Sendai, Japan, Oct. 2007. in press.

J Schmidhuber. Self-delimiting neural networks. *arXiv preprint arXiv:1210.0118*, 2012.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. doi: 10.1016/j.neunet.2014.09.003. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

J. Schmidhuber and R. Huber. Learning to Generate Artificial Fovea Trajectories for Target Detection. *International Journal of Neural Systems*, 2(1):135–141, 1991.

J. Schmidhuber, S. Hochreiter, and Y. Bengio. Evaluating benchmark problems by random guessing. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001. Based on TR IDSIA-19-96 by Schmidhuber and Hochreiter (1996).

F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber. Parameter-Exploring Policy Gradients. *Neural Networks*, 23(4):551–559, 2010.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy Gradients with Parameter-Based Exploration for Control. In *Proceedings of the International Conference on Artificial Neural Networks ICANN*, 2008a.

Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. Parametric Policy Gradients for Robotics. In *1st International Workshop on Cognition for Technical Systems*, 2008b.

Guy Shani and Ronen I Brafman. Resolving perceptual aliasing in the presence of noisy sensors. In *Advances in Neural Information Processing Systems*, pages 1249–1256, 2004.

John Andrew Simpson and Edmund S. Weiner. *The Oxford English Dictionary*, volume 2. Clarendon Press Oxford, 1989.

S. Singh, A.G. Barto, and N. Chentanez. Intrinsically Motivated Reinforcement Learning. *Advances in Neural Information Processing Systems*, 17:1281–1288, 2005.

Rupesh Kumar Srivastava, Bas R Steunebrink, Marijn Stollenga, and Jurgen Schmidhuber. Continually adding self-invented problems to the repertoire: first experiments with powerplay. In *Development and Learning and Epigenetic Robotics (ICDL), 2012 IEEE International Conference on*, pages 1–6. IEEE, 2012.

Rupesh Kumar Srivastava, Bas R Steunebrink, and Jürgen Schmidhuber. First experiments with powerplay. *Neural Networks*, 2013.

H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity*, volume 79. Birkhäuser, 1994.

M. Sumner, E. Frank, and M. Hall. Speeding Up Logistic Model Tree Induction. *Knowledge Discovery in Databases: PKDD 2005*, pages 675–683, 2005.

Ron Sun and C Lee Giles. *Sequence learning: Paradigms, algorithms, and applications*, volume 1828. Springer, 2001.

R. Sutton. Open Theoretical Questions in Reinforcement Learning. In *Computational Learning Theory*, pages 637–638. Springer, 1999.

Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ML*, pages 216–224, 1990.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, 2000.

R.S. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. *Advances in Neural Information Processing Systems*, pages 1038–1044, 1996. ISSN 1049-5258.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

G. Tesauro. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural computation*, 6(2):215–219, 1994. ISSN 0899-7667.

S. Thrun and A. Schwartz. Issues in Using Function Approximation for Reinforcement Learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

S.B. Thrun. The Role of Exploration in Learning Control. *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pages 527–559, 1992.

S. Timmer and M. Riedmiller. Fitted Q-Iteration with CMACs. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 1–8. IEEE, 2007.

Holger Urbanek, Alin Albu-Schäffer, and Patrick van der Smagt. Learning from demonstration: repetitive movements for autonomous service robotics. In *Proceedings of Intelligent Robots and Systems, 2004.*, volume 4, pages 3495–3500. IEEE, 2004.

H. van Hasselt and M. Wiering. Reinforcement Learning in Continuous Action Spaces. In *Proc. IEEE Symp. Approx. Dynamic Programming and Reinforcement Learning*, volume 272, page 279, 2007.

Cornelis J Van Rijsbergen, Stephen Edward Robertson, and Martin F Porter. *New Models in Probabilistic Information Retrieval*. Computer Laboratory, University of Cambridge, 1980.

S. Vijayakumar and S. Schaal. Locally Weighted Projection Regression: An O (n) Algorithm for Incremental Real Time Learning in High Dimensional Space. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, volume 1, pages 288–293, 2000.

C.J.C.H. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8(3):279–292, 1992.

P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.

P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, pages 762–770, 1981.

Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

Steven D Whitehead and Dana H Ballard. Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83, 1991.

B. Widrow and M.E. Hoff. Associative storage and retrieval of digital information in networks of adaptive neurons. *Biological Prototypes and Synthetic Systems*, 1:160, 1962.

M. Wiering and J. Schmidhuber. Efficient Model-Based Exploration. *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, 1998.

D. Wierstra and J. Schmidhuber. Policy Gradient Critics. In *Proceedings of the Eighteenth European Conference on Machine Learning ECML*, 2007.

Daan Wierstra, Alexander Foerster, Jan Peters, and Juergen Schmidhuber. Solving deep memory POMDPs with recurrent policy gradients. In *Proceedings of the International Conference of Artificial Neural Networks (ICANN)*, 2007.

R.J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural computation*, 1(2):270–280, 1989.

Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8:229–256, 1992.

H. Wold. *Path Models with Latent Variables: The NIPALS Approach*. Acad. Press, 1975.

Xindong Wu, Kui Yu, Hao Wang, and Wei Ding. Online Streaming Feature Selection. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1159–1166, 2010.

Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120, 1995.