

# Advancing the Applicability of Reinforcement Learning to Autonomous Control

## Dissertation

zur Erlangung des akademischen Grades  
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung  
der Technischen Universität Ilmenau

von Dipl.-Inf. Alexander Hans  
geboren am 18.06.1981 in Nordhausen

Tag der Einreichung: 02.12.2013  
Tag der Verteidigung: 15.07.2014

1. Gutachter: Prof. Dr. Horst-Michael Groß  
2. Gutachter: Prof. Dr. Damien Ernst  
3. Gutachter: Dr. Hans-Georg Zimmermann



# Abstract

With data-efficient reinforcement learning (RL) methods impressive results could be achieved, e.g., in the context of gas turbine control. However, in practice the application of RL still requires much human intervention, which hinders the application of RL to autonomous control. This thesis addresses some of the remaining problems, particularly regarding the reliability of the policy generation process.

The thesis first discusses RL problems with discrete state and action spaces. In that context, often an MDP is estimated from observations. It is described how to incorporate the estimators' uncertainties into the policy generation process. This information can then be used to reduce the risk of obtaining a poor policy due to flawed MDP estimates. Moreover, it is discussed how to use the knowledge of uncertainty for efficient exploration and the assessment of policy quality without requiring the policy's execution.

The thesis then moves on to continuous state problems and focuses on methods based on fitted Q-iteration (FQI), particularly neural fitted Q-iteration (NFQ). Although NFQ has proven to be very data-efficient, it is not as reliable as required for autonomous control. The thesis proposes to use ensembles to increase reliability. Several ways of ensemble usage in an NFQ context are discussed and evaluated on a number of benchmark domains. It shows that in all considered domains with ensembles good policies can be produced more reliably.

Next, policy assessment in continuous domains is discussed. The thesis proposes to use fitted policy evaluation (FPE), an adaptation of FQI to policy evaluation, combined with a different function approximator and/or different dataset to obtain a measure for policy quality. Results of experiments show that extra-tree FPE, applied to policies generated by NFQ, produces value functions that can well be used to reason about the true policy quality.

Finally, the thesis combines ensembles and policy assessment to derive methods that can deal with changing environments. The major contribution is the evolving ensemble. The policy of the evolving ensemble changes slowly as new policies are added and old policies removed. It turns out that the evolving ensemble approaches work considerably better than simpler approaches like single policies learned with recent observations or simple ensembles.



# Zusammenfassung

Mit dateneffizientem Reinforcement Learning (RL) konnten beeindruckende Ergebnisse erzielt werden, z.B. für die Regelung von Gasturbinen. In der Praxis erfordert die Anwendung von RL jedoch noch viel manuelle Arbeit, was bisher RL für die autonome Regelung untauglich erscheinen ließ. Die vorliegende Arbeit adressiert einige der verbleibenden Probleme, insbesondere in Bezug auf die Zuverlässigkeit der Policy-Erstellung.

Es werden zunächst RL-Probleme mit diskreten Zustands- und Aktionsräumen betrachtet. Für solche Probleme wird häufig ein MDP aus Beobachtungen geschätzt, um dann auf Basis dieser MDP-Schätzung eine Policy abzuleiten. Die Arbeit beschreibt, wie die Schätzer-Unsicherheit des MDP in die Policy-Erstellung eingebracht werden kann, um mit diesem Wissen das Risiko einer schlechten Policy aufgrund einer fehlerhaften MDP-Schätzung zu verringern. Außerdem wird so effiziente Exploration sowie Policy-Bewertung ermöglicht.

Anschließend wendet sich die Arbeit Problemen mit kontinuierlichen Zustandsräumen zu und konzentriert sich auf RL-Verfahren, welche auf Fitted Q-Iteration (FQI) basieren, insbesondere Neural Fitted Q-Iteration (NFQ). Zwar ist NFQ sehr dateneffizient, jedoch nicht so zuverlässig, wie für die autonome Regelung nötig wäre. Die Arbeit schlägt die Verwendung von Ensembles vor, um die Zuverlässigkeit von NFQ zu erhöhen. Es werden eine Reihe von Möglichkeiten der Ensemble-Nutzung entworfen und evaluiert. Bei allen betrachteten RL-Problemen sorgen Ensembles für eine zuverlässigere Erstellung guter Policies.

Im nächsten Schritt werden Möglichkeiten der Policy-Bewertung bei kontinuierlichen Zustandsräumen besprochen. Die Arbeit schlägt vor, Fitted Policy Evaluation (FPE), eine Variante von FQI für Policy Evaluation, mit anderen Regressionsverfahren und/oder anderen Datensätzen zu kombinieren, um ein Maß für die Policy-Qualität zu erhalten. Experimente zeigen, dass Extra-Tree-FPE ein realistisches Qualitätsmaß für NFQ-generierte Policies liefern kann.

Schließlich kombiniert die Arbeit Ensembles und Policy-Bewertung, um mit sich ändernden RL-Problemen umzugehen. Der wesentliche Beitrag ist das Evolving Ensemble, dessen Policy sich langsam ändert, indem alte, untaugliche Policies entfernt und neue hinzugefügt werden. Es zeigt sich, dass das Evolving Ensemble deutlich besser funktioniert als einfachere Ansätze.



# Danksagung

Die vorliegende Arbeit entstand während meiner Zeit als Doktorand bei der Siemens AG, Corporate Technology, Intelligent Systems and Control in München. Gleichzeitig war ich externer Doktorand des Fachgebiets für Neuroinformatik und Kognitive Robotik der TU Ilmenau.

Zunächst danke ich Prof. Dr. Horst-Michael Groß für die Bereitschaft, die universitäre Betreuung der Arbeit zu übernehmen, und die damit verbundene Unterstützung. Weiterhin danke ich Prof. Dr. Thomas Runkler für die Möglichkeit, diese Arbeit als Mitglied seiner ausgezeichneten Fachgruppe anfertigen zu können.

Mein besonderer Dank gilt Dr. Steffen Udluft, der mir als Betreuer nahezu täglich mit Rat und Tat zur Seite stand. Von unseren zahlreichen Diskussionen profitierte diese Arbeit sehr. Häufig wurden so aus ersten, vagen Ideen umsetzbare Konzepte. Ich habe von ihm sehr viel gelernt und unsere Zusammenarbeit sehr genossen. Weitere Mitglieder der Fachgruppe, insbesondere Volkmar Sterzing, Dr. Ralph Grothmann und Dr. Hans-Georg Zimmermann, haben mir nicht nur zum Thema der Dissertation wertvolle Hinweise gegeben. Letztgenanntem danke ich darüber hinaus für die Bereitschaft die Rolle eines Gutachters zu übernehmen. Von Dieter Bogdoll und Dr. Christoph Tietz habe ich viel über gute Software-Entwicklung gelernt. Auch geht mein Dank an meine Mit-Doktoranden Dr. Daniel Schneegaß, Dr. Anton Maximilian Schäfer sowie in besonderem Maße Sigmund Düll. Auch Diskussionen mit ihnen haben die Arbeit nicht unerheblich beeinflusst. Sie alle haben zu einem ausgesprochen angenehmen Arbeitsklima beigetragen, das jederzeit von Offenheit und Freundlichkeit geprägt war.

Dr. Damien Ernst danke ich für seine Einsichten in Fitted Q-Iteration und Extra-Trees sowie für seine Bereitschaft als Zweitgutachter zu fungieren.

Schließlich wäre all dies nicht möglich gewesen ohne die bedingungslose Unterstützung, die ich von meiner Familie und insbesondere meinen Eltern immer erfahren habe. Bei all meinen Vorhaben konnte ich mich stets auf sie verlassen.

Vor allem aber möchte ich mich bei Anne bedanken, die mit mir viele Momente des Erfolgs, aber auch der Frustration geteilt hat und in der Lage war, mich stets zur Weiterarbeit zu motivieren.



# Contents

<b>List of Figures</b>	<b>xii</b>
<b>List of Tables</b>	<b>xiii</b>
<b>List of Acronyms</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 RL for Autonomous Learning . . . . .	3
1.2 Incorporation of Uncertainty . . . . .	5
1.3 Ensembles for More Reliable Neural RL . . . . .	6
1.4 Real-World Example: Gas Turbine Control . . . . .	6
1.5 Outline . . . . .	9
1.6 Publications . . . . .	10
<b>2 Reinforcement Learning</b>	<b>13</b>
2.1 Overview . . . . .	13
2.2 Markov Decision Processes . . . . .	14
2.3 Value Functions . . . . .	15
2.4 Dynamic Programming . . . . .	18
2.4.1 Policy Evaluation, Improvement, and Iteration . . . . .	19
2.4.2 Value Iteration . . . . .	20
2.4.3 Learning from Observations . . . . .	20
2.5 Temporal-Difference Learning . . . . .	22
2.6 Data-Efficiency . . . . .	23
2.7 Function Approximation . . . . .	25
2.8 Policy Gradient and Policy Search Methods . . . . .	27
2.9 Exploration . . . . .	28
2.10 Further Reading . . . . .	30
<b>3 Uncertainty Awareness in Discrete Domains</b>	<b>31</b>
3.1 Basic Idea . . . . .	31
3.2 Estimators . . . . .	32

3.2.1	Frequentist Estimate . . . . .	33
3.2.2	Bayesian Estimate . . . . .	34
3.3	Monte Carlo Uncertainty Estimate . . . . .	35
3.4	Uncertainty-Aware Value Iteration . . . . .	38
3.5	Full-Matrix Uncertainty Propagation . . . . .	39
3.6	Efficient Diagonal Approximation . . . . .	41
3.7	Summary . . . . .	44
<b>4</b>	<b>Discrete Domain Applications of Uncertainty Awareness</b>	<b>45</b>
4.1	Quality Assurance . . . . .	45
4.1.1	Example . . . . .	47
4.1.2	Benchmarks . . . . .	50
4.1.3	Experiments and Results . . . . .	52
4.2	Self-Assessment . . . . .	54
4.2.1	Value Function-Based Self-Assessment . . . . .	55
4.2.2	Experiments . . . . .	56
4.2.3	Related Work . . . . .	61
4.2.4	Conclusion . . . . .	62
4.3	Exploration . . . . .	63
4.3.1	Related Work . . . . .	63
4.3.2	Benchmarks . . . . .	65
4.3.3	Results and Discussion . . . . .	66
4.4	Summary . . . . .	68
<b>5</b>	<b>Ensembles for More Reliable Policy Identification</b>	<b>71</b>
5.1	Neural Fitted Q-Iteration . . . . .	71
5.1.1	Fitted Q-Iteration . . . . .	72
5.1.2	Problems of NFQ . . . . .	72
5.2	Ensemble Methods . . . . .	78
5.3	Ensembles in Reinforcement Learning . . . . .	80
5.4	Experiments . . . . .	82
5.4.1	Cart-Pole . . . . .	82
5.4.2	Pole-Balancing . . . . .	83
5.4.3	Wet-Chicken . . . . .	84

---

5.4.4	Experimental Setup . . . . .	84
5.4.5	Results . . . . .	85
5.5	Why Do Ensembles of NFQ Policies Work? . . . . .	90
5.6	Continuous Actions . . . . .	91
5.7	Summary and Conclusion . . . . .	93
<b>6</b>	<b>Self-Assessment in Continuous Domains</b>	<b>95</b>
6.1	Value Function-Based Self-Assessment . . . . .	95
6.2	Fitted Policy Evaluation . . . . .	96
6.3	Correlation Between True Performance and Value Function Estimate	98
6.4	Different Function Approximator and Different Data . . . . .	99
6.5	Policy Selection and Rejection . . . . .	101
6.6	Weighted Ensembles . . . . .	103
6.7	Summary . . . . .	106
<b>7</b>	<b>Autonomous Control in Changing Environments</b>	<b>109</b>
7.1	Policy Selection in Changing Environments . . . . .	110
7.2	The Evolving Ensemble . . . . .	112
7.2.1	Idea . . . . .	112
7.2.2	Experiments . . . . .	113
7.2.3	Conclusions . . . . .	122
7.3	Summary . . . . .	123
<b>8</b>	<b>Conclusion</b>	<b>125</b>
8.1	Summary . . . . .	125
8.2	Contributions . . . . .	129
8.3	Future Research . . . . .	132
	<b>Bibliography</b>	<b>135</b>
	<b>Index</b>	<b>145</b>



# List of Figures

1.1	Components and their interaction for RL-based gas turbine control	8
2.1	Interaction between agent and environment . . . . .	14
3.1	Exemplary return distributions of two actions . . . . .	37
4.1	Simple three-state MDP . . . . .	46
4.2	Histograms of $Q$ -values of the three-state MDP . . . . .	50
4.3	Visualization of the archery benchmark . . . . .	52
4.4	Performance of policies generated with standard value iteration and uncertainty aware approaches . . . . .	52
4.5	Histograms of mean rewards and mean and quantile rewards . . .	53
4.6	Results for the archery domain . . . . .	54
4.7	Policy ranking results for the archery benchmark . . . . .	58
4.8	Histograms of $J(\pi)$ and $J_u(\pi)$ . . . . .	59
4.9	Policy ranking results for the wet-chicken benchmark . . . . .	60
4.10	Trap domain . . . . .	61
4.11	Policy ranking results for the trap domain . . . . .	62
4.12	River-swim domain . . . . .	65
4.13	Cumulative rewards . . . . .	68
4.14	Immediate rewards . . . . .	69
5.1	Performances of policies from repeated NFQ runs . . . . .	73
5.2	Maximum $Q$ -values with different activation functions . . . . .	75
5.3	Illustration of the cart-pole benchmark. . . . .	84
5.4	Deep, cascaded neural network . . . . .	85
5.5	Histogram of majority ratios . . . . .	91
5.6	Example of superimposition of Gaussians . . . . .	93
6.1	Performance of selected/rejected policies (pole-balancing, 50 epi.)	102
6.2	Performance of selected/rejected policies (pole-balancing, 100 epi.)	103
6.3	Performance of selected/rejected policies (wet-chicken) . . . . .	103
7.1	Performance of equally and TreeFPE-weighted ensembles . . . . .	111

---

7.2	Setup of the evolving ensemble experiment . . . . .	114
7.3	Evolving ensemble results from the pole-balancing benchmark . .	116
7.4	Histogram of the duration of policy usage . . . . .	118
7.5	Evolving ensemble results from the cart-pole benchmark . . . . .	119
7.6	Results from the randomly-replacing evolving ensemble . . . . .	121

# List of Tables

3.1	Time and space complexities of the algorithms. . . . .	44
4.1	Exemplary observations of the simple three-state MDP. . . . .	47
4.2	$Q$ -values for the expectation-optimal policy . . . . .	48
4.3	$Q$ -values and standard deviation using the sampling approach . .	48
4.4	$Q$ -values and standard deviation using the sampling approach with $\xi = 1$ . . . . .	50
4.5	Best results obtained using the various algorithms in the river-swim and trap domains . . . . .	66
4.6	Parameters used for the experiments . . . . .	66
4.7	Computation times . . . . .	67
5.1	Number of successful NFQ policies with different activation functions	76
5.2	Complexity of ensemble methods in NFQ . . . . .	82
5.3	Parameters used for the cart-pole and pole-balancing benchmarks	83
5.4	Results of majority voting for the pole-balancing benchmark . . .	86
5.5	Results of Q-averaging for the pole-balancing benchmark . . . . .	87
5.6	Results of “most agreeable” policies for the pole-balancing benchmark . . . . .	87
5.7	Results of majority voting with policies from successive iterations for the pole balancing benchmark . . . . .	88
5.8	Experimental results using policies from a single NFQ run and different ensemble policies . . . . .	89
6.1	Correlations between true policy performance and value functions	99
6.2	Correlations between true policy performance and $\hat{J}^{\mu, S_0}$ . . . . .	100
6.3	Performance of differently weighted ensembles . . . . .	104
6.4	Performances of policies from the “most agreeable” approach and “most preferable” policies . . . . .	105



# List of Acronyms

ADP	approximate dynamic programming
DP	dynamic programming
DUIPI	diagonal approximation of uncertainty-incorporating policy iteration
DUIPI-QM	diagonal approximation of uncertainty-incorporating policy iteration with $Q$ modification
FPE	fitted policy evaluation
FQI	fitted $Q$ -iteration
MDP	Markov decision process
MSE	mean squared error
NFPE	neural fitted policy evaluation
NFQ	neural fitted $Q$ -iteration
POMDP	partially observable Markov decision process
RL	reinforcement learning
TD	temporal-difference
UP	uncertainty propagation



# 1

## Introduction

Reinforcement learning (RL) (Sutton and Barto, 1998) has become an attractive option for solving challenging problems of optimal control, like controlling a gas turbine. It is a type of machine learning concerned with the interaction with some system. Often the notion of an agent interacting with an external environment is used. The agent can observe the state of the environment and influence it by carrying out actions. An action leads to a state transition and a new state, which again the agent can observe and base its next action on. Along with the next state a scalar value, the reward, is given. The agent's aim is to find and follow a policy that chooses those actions that maximize the reward in the long run.

RL is interesting for industrial problems of optimal control, because it learns from data—actual observations from the system to be controlled. Learning from data offers two major advantages. First, such a method can be applied even when no or only an insufficient analytical description of the system is available. Second, the method can automatically adapt the control strategy to changing characteristics of the system, e.g., introduced by wear.

In machine learning RL lies between supervised and unsupervised learning. While in supervised learning the user gives input-target pairs and wants the machine to learn a mapping from input to target, in unsupervised learning one is interested in finding structures in the data without specifying a target (e.g., Hastie, Tibshirani, and Friedman, 2001). Using the reward function the user can specify the objective in an RL problem. However, the policy, i.e., the mapping from states to actions, maximizing the long-term reward is not given in advance. Finding it is the essence of RL, especially when knowledge about the environment is limited and only given in the form of observations consisting of state transitions and rewards (i.e., tuples consisting of state, action, successor state, and reward).

If the dynamics of the environment, i.e., the probability distribution over successor states, and the reward function are known, dynamic programming (DP) (Bellman, 1957) can be used to determine an optimal policy. If because of a continuous state space or a large number of discrete states the policy cannot be stored tabularly (i.e., keeping for each possible state the optimal action in memory), one must employ function approximation and therefore use approximate dynamic programming (ADP) (Buşoniu, Babuška, Schutter, and Ernst, 2010).

With an incomplete knowledge of the environment (A)DP alone is not applicable and can only be part of an RL solution that first estimates a model of the environment and then uses (A)DP to determine the optimal policy w.r.t. that model.

The environment is usually assumed to be a Markov decision process (MDP) (Puterman, 1994)  $M := (S, A, P, R)$ , where  $S$  is the state space,  $A$  the action space,  $P : S \times A \times S \mapsto [0, 1]$  the transition probabilities, and  $R : S \times A \times S \mapsto R$  the reward function. An important feature of MDPs is the Markov property—given an MDP  $M$  the knowledge of the current state and action is sufficient to determine (the probability for) the next state. No historical information of previous states is required.

In principle, RL can solve any problem that can be formulated as an MDP. However, formulating the problem as an MDP is not sufficient to actually find a suitable solution, it is merely one of the first steps when approaching a problem using RL.

Problems that are well-suited to be formulated as an MDP include optimal control problems. In optimal control, one can usually observe the state of the system under control and influence it through control actions. The optimization goal is formulated using the reward function. Games as well are often easily formulated as MDPs. For example, consider the game of chess. The state would be comprised of the current board configuration, possible actions would include all valid moves, a reward of 0 would be given for all actions except the last one before the end of the game: if the agent wins, it receives reward 1; if it loses, a reward of  $-1$  is given; for a draw it receives 0. Although only the very last action receives a direct reward, all previous actions influenced the way that led to the final board configuration. In theory, an RL agent is able to learn from experience the moves that lead to winning the game. In practice, however, a naïve RL solution for the game of chess will not be possible, since the number of possible board positions is prohibitively large.<sup>1</sup>

Nonetheless, there are quite a few “success stories” of RL, including games. Arguably the most famous application of RL to games is Tesauro’s *Td-Gammon* (Tesauro, 1994), an RL program that learned to play backgammon at master level by repeatedly playing against itself. While the state space of backgammon is not as large as that of chess, it is still far too large to use a tabular representation. Tesauro used a neural network as function approximator and a learning algorithm called *Q-learning* (Watkins, 1989) to train it. There are a number of reports of successful application of RL to real-world problems from the robotics domain (e.g., Merke and Riedmiller, 2001; Abbeel, Coates, Quigley, and Ng, 2006; Lee et al., 2006; Peters and Schaal, 2008; Kober and Peters, 2012) as well as optimal control of combustion processes (Stephan et al., 2000) and gas turbine control (Schäfer, Schneegaß, Sterzing, and Udluft, 2007).

---

<sup>1</sup>According to an early estimate by Shannon (1950) the number of possible positions is roughly  $10^{43}$ , recently an upper bound of  $10^{46.7}$  was proven (Tromp, 2010).

An important prerequisite to make RL applicable to interesting real-world problems is *data-efficiency*, i.e., requiring only few observations to derive a near-optimal policy. This is necessary as observations of a real plant are usually expensive—it takes time to generate them and the disruption of the plant’s regular operation should be kept as low as possible. While Tesauro’s Td-Gammon was not particularly data-efficient—it required several hundreds of thousands games—the issue has been addressed in recent years by RL methods that reformulate the task of learning an optimal policy from a set of observations to a series of regression problems and then applying powerful function approximators like neural networks to these regression problems (e.g., Riedmiller, 2005; Schneegaß, Udluft, and Martinetz, 2006; Schneegaß, Udluft, and Martinetz, 2007a).

## 1.1 RL for Autonomous Learning

The focus of this thesis is making RL methods more autonomous. This means requiring less human intervention when applying RL to a problem, but it mostly means enabling the RL agent to run without (or only very little) human intervention once the basic steps of RL application are done. Of course, this includes controlling a system using RL in closed loop, but even more importantly it also includes generating new policies online and thus adapting to changing characteristics of the environment.

Although in principle RL should be applicable for autonomous learning, i.e., the user only formulates the problem as an MDP and RL takes care of everything else and finally returns a (near-optimal) policy, in practice it is not that easy. In particular, after having formulated the problem as an MDP, the application of RL requires the following steps.

1. *Analysis of the problem (MDP)*: As a first step one must analyze the type of MDP. Are the state and actions spaces discrete or continuous? If discrete, how many states and actions must be considered? Are the state transitions and reward function stochastic or deterministic?
2. *Choice of RL algorithm*: Based on the analysis of the previous step, a suitable RL algorithm must be chosen. One could use model-based RL, where first a model of the environment is estimated and then (approximate) dynamic programming used to find a policy that is optimal w.r.t. that model. On the other hand, there are model-free approaches. Likewise, depending on whether either or both, the state space and action space, are discrete or continuous and the number of states and actions one must decide for or against using function approximation. If function approximation should be used, what is a suitable method? Neural networks, kernel-based approaches, regression trees?

3. *Choice of meta parameters of the algorithm:* Many algorithms have certain meta parameters. Those are not parameters determined during the process of learning like the weights of a neural network, but parameters like its topology (e.g., number of hidden layers, number of neurons in a hidden layer), the learning algorithm (e.g., back propagation) and the learning rate; for a kernel-based method one has to choose the type of kernel and the so-called hyper-parameters of the kernel.
4. *Monitoring of the learning process:* For many learning algorithms there are indicators that can be used to monitor the learning process. For instance, one would observe the error curves of a neural network for a training and validation set to determine whether learning is successful and to be able to detect overfitting.
5. *Evaluation of the result (policy):* Once a policy has been generated, the policy must be evaluated to decide whether it is good enough. If not, maybe more observations are necessary, or one has to go back to one of the previous steps and check the correctness of the MDP evaluation, consider other algorithms, or tune meta parameters, and finally re-run the algorithm, possibly with more data. The easiest way of evaluating a policy is to actually run it using the actual MDP. However, if the MDP is a real system like a gas turbine and not just a simulation running on a computer, this can be very costly. Therefore, a measure of policy performance without the need for running the policy on the actual problem is desirable.

Looking at the above enumeration, it becomes obvious that currently RL is far from being truly autonomous. In addition to the decisions the user has to make in terms of algorithm and parameter selection, often it is also required to monitor the learning process and assess final policies before actually using them for the system to be controlled. So even when steps 1–3 are completed, it is in general not possible to apply the resulting algorithm to a system and let it run on its own in closed loop. The learning process still needs to be monitored to ensure proper learning. Moreover, it is in general not possible to guarantee a certain quality of the resulting policy. For the application of RL to an autonomously learning controller for a technical system, however, the proper function of the learning controller must be ensured.

The aim of this thesis is to advance the applicability of RL to problems of autonomous control. While potentially all items of the enumeration above could be addressed, the first two points will be left out. They can normally be taken care of using prior knowledge about the problem at hand; this knowledge should be incorporated where possible. For instance, the user knows whether the problem's state space is discrete or continuous and can therefore choose an appropriate algorithm. Moreover, the first two steps usually have to be addressed only once for a given application. For example, when applying RL to gas turbine control, an algorithm that works well for one turbine will most probably deliver good

results for a similar gas turbine as well. On the other hand, parts of point 3 (choice of meta parameters), and points 4 (monitoring the learning process) and 5 (evaluation of the policy) need to be done repeatedly if an RL method is to be applied to similar systems or used to continuously adapt a policy to a changing environment.

There are two major areas of contributions towards autonomous RL. On the one hand, the thesis tries to make existing methods more robust, so they are less sensitive to parameter choices and produce good results more reliably. In discrete domains, uncertainty is incorporated to reduce the probability of determining poor policies. In continuous domains this is achieved through ensembles. On the other hand, methods are discussed that allow for the assessment of a policy's quality without actually executing it. Again, the knowledge of uncertainty helps in discrete domains. For continuous domains the combination of policy evaluation and ensembles and the use of different datasets and function approximators for policy assessment are proposed.

## 1.2 Incorporation of Uncertainty

As a first step, the thesis deals with ways to incorporate uncertainty into methods for discrete state and action spaces. In particular, this means applying uncertainty propagation, a common concept in statistics (see, e.g., D'Agostini, 2003), to policy iteration to obtain the uncertainty of the so-called *Q-function*, a mapping from a state-action pair  $(s, a)$  to a scalar value that determines the quality of  $(s, a)$ . Policy iteration is based on the Bellman optimality equation and can be used to determine the *Q-function*  $Q^*$  of the optimal policy  $\pi^*$  (Bellman, 1957), where  $\pi^*(s) := \arg \max_a Q^*(s, a)$ . Uncertainty propagation for policy iteration was first introduced by Schneegaß, Udluft, and Martinetz (2008) for quality assurance—by avoiding state-action pairs with high uncertainty it is possible to decrease the probability of obtaining an insufficient policy. Unfortunately, the computational burden of that method is very high and becomes prohibitive as the number of the MDP's states grows significantly beyond 100. This is mainly due to the need of maintaining a covariance matrix, whose size grows in  $O(|S|^5|A|^3)$  with the number of states  $|S|$  and number of actions  $|A|$ . In this thesis, an approximate method for uncertainty propagation is proposed. By considering only the diagonal elements of the covariance matrix, the complexity of the method is reduced and remains the same as that of regular policy iteration. It is shown that the approximation can still be successfully used to reduce the probability of obtaining a poor policy. This way the uncertainty is used to derive good policies more reliably and therefore helps the autonomous agent to become more robust.

Moreover, the knowledge of uncertainty can be used to more reliably assess a policy's quality by looking at its *Q-function*. The parameters of the true MDP are usually unknown and must be estimated from observations. A *Q-function*

determined w.r.t. to the estimated MDP only gives expected long-term rewards for the estimated MDP; the results can be misleading w.r.t. the real MDP. Using the uncertainty it is possible to correct the policy quality estimate.

### 1.3 Ensembles for More Reliable Neural RL

Many interesting problems exhibit a continuous state space instead of a discrete one. When dealing with continuous state spaces, one has to introduce some sort of function approximation. In this thesis, mostly neural networks are used as such function approximators, since they have excellent generalization properties (Hastie, Tibshirani, and Friedman, 2001, p. 351). They are employed for fitted  $Q$ -iteration (FQI), where the task of learning the  $Q$ -function is reformulated as a set of regression problems (Ernst, Geurts, and Wehenkel, 2005). Although various impressive results have been reported using FQI with neural networks, then called neural fitted  $Q$ -iteration (NFQ) (Riedmiller, 2005), a number of problems have been reported as well (Thrun and Schwartz, 1993; Gordon, 2001; Gaskett, 2002; Gabel and Riedmiller, 2006). This leads to NFQ approaches being not as reliable as would be desirable for autonomous RL. One way of dealing with this and producing successful policies more reliably is the usage of ensembles. In supervised learning, ensembles of independently trained learners have been used successfully to produce better and more reliable results than when using a single learner, both for regression and classification (Dietterich, 2000). RL can also benefit from ensembles. For instance, instead of learning only a single policy given a set of observations, one can learn multiple policies and combine them to a final policy (e.g., using majority voting). The present work proposes a number of ways of using ensembles in an NFQ context and shows empirically that ensembles indeed lead to more reliable policy identification.

### 1.4 Real-World Example: Gas Turbine Control

To illustrate the application of RL to a complex technical real-world problem and some of the difficulties, consider the problem of optimal control of gas turbines. In gas turbine control, the aim is to operate the turbine in a way that gives the desired power output while at the same time keeping polluting emissions (CO, NO<sub>x</sub>) and vibrations caused by combustion dynamics low. The gas is injected through multiple burners. At each burner, the fuel flow is split into different fractions enabling control over the combustion process. Traditionally, turbine operators use tables and their experience to determine the optimal working point settings, i.e., the different fractions, for a given situation, depending on parameters such as the desired power output and ambient influences like air temperature and pressure. Low-level controllers take care of reaching and keeping the currently set working point. As those low-level controllers already work optimally, RL is used

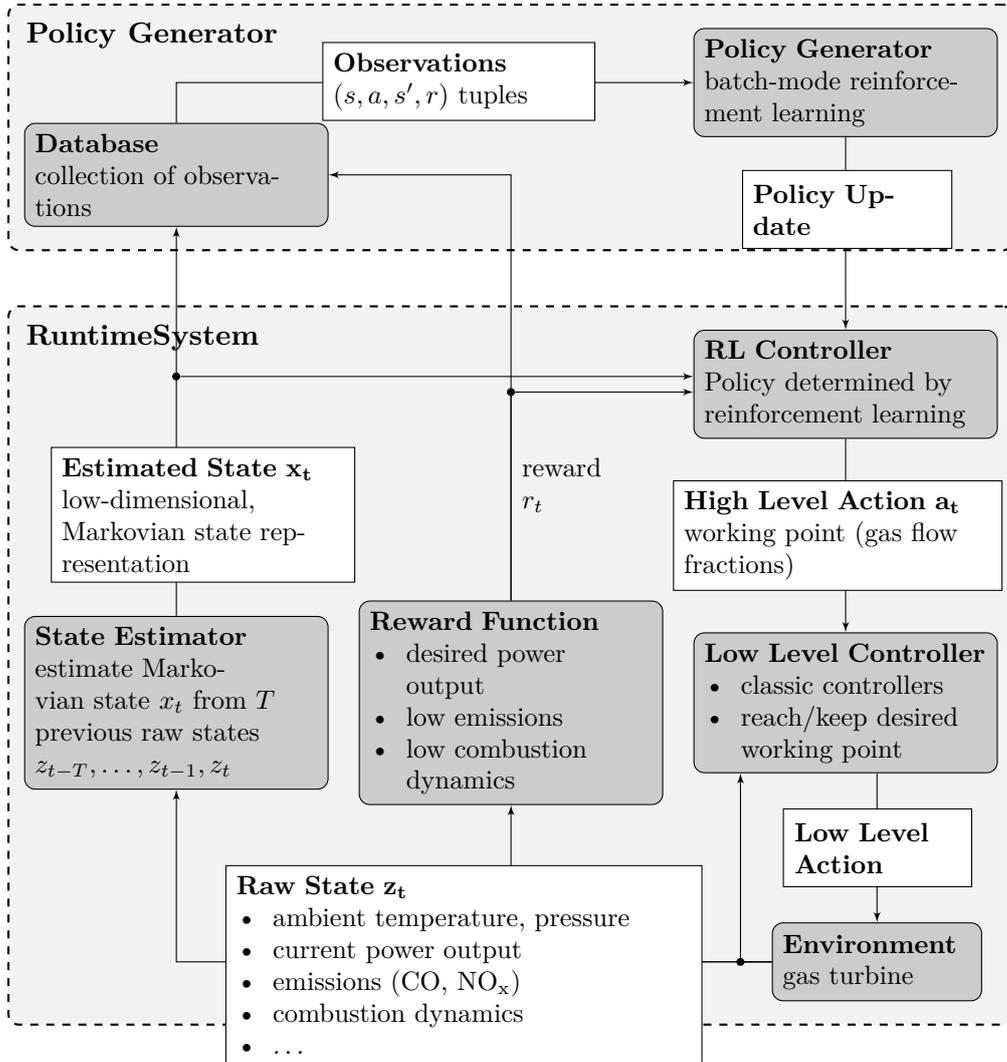
to optimize the high-level control. It is advantageous to use RL because one can consider more parameters than covered by the tables, therefore giving working points that are close to optimality for a given situation. Moreover, during the lifetime of a turbine, an RL controller can learn new policies using actual observations of the turbine. It can therefore adapt the control strategy to changing turbine characteristics, e.g., introduced by wear.

Consider the enumeration of the steps required when applying RL again (Section 1.1). Performing those steps manually is feasible when one is interested in generating a single policy that is then applied to the system. If, however, not just the policy, but also the learning process is to be run in closed loop, the learning process must be sufficiently robust to produce good policies reliably.

Figure 1.1 shows the components and their interaction during runtime. The RL controller contains the policy that gives the (high level) action (working point) for the current situation. This action is passed to a low level controller that takes care of reaching and keeping that working point. The low level controller influences the turbine through low level actions and works in a closed loop by monitoring the current raw state of the turbine (measurements such as ambient air temperature and pressure, current power output, emissions, or combustion dynamics). Furthermore, the raw state is passed to the reward function and a state estimator. The reward function calculates a scalar reward value from (parts of) the current state. Since the process described by a series of raw states is usually non-Markovian, i.e., the raw state from one time step is not sufficient to completely describe the state of the environment, requiring data from a number of past time steps, a state estimator is used to estimate a Markovian state representation. The observations of the turbine are stored in a database.

The low level controller and the turbine constitute the conventional setup. For the RL solution, the RL controller, the reward function, and the state estimator are added. Then the working points are not defined by static tables any more, but instead given by the current policy of the RL controller. The RL controller's policy is updated by the policy generator once it has determined a new policy. In this setting, the RL agent consists of two parts: the online agent executing the policy (RL controller) and the offline agent generating new policies (policy generator). To enable autonomous operation of a system like that, the following requirements must be met:

- The policy generator must be able to autonomously determine a new policy. Manual monitoring of the learning process must not be required. Furthermore, once the learning algorithm's parameters are set, it should not be necessary to adjust them manually.
- A high quality of the RL controller's policy must be maintained. Furthermore, differences between subsequent policies should be small, so that a smooth transition from one policy to the next is possible.



**Figure 1.1:** Components and their interaction for RL-based gas turbine control. Components are depicted as gray rectangles with rounded corners, white rectangles describe data being passed from one component to another.

## 1.5 Outline

The remainder of the thesis is structured as follows:

- **Chapter 2** gives a more detailed introduction to and overview of RL.
- **Chapter 3** details methods for uncertainty awareness in discrete domains. The basic idea is described, i.e., to determine the  $Q$ -function's uncertainty based on the uncertainty of the MDP estimate. To this end possible choices of estimators are presented. Then a Monte Carlo method to estimate the uncertainty is described, followed by a direct method using uncertainty propagation. Since this method's computational burden is high, an efficient approximation is proposed.
- **Chapter 4** discusses possible applications of uncertainty awareness in discrete domains. The knowledge of uncertainty can be used in a quality assurance setting, where one wants to prevent the generation of very poor policies. Further applications are self-assessment, i.e., determining the actual quality of a policy using the  $Q$ -function and its uncertainty, and uncertainty-based exploration. For all applications a number of experiments are performed to evaluate the methods.
- Starting with **Chapter 5** the focus shifts from discrete to continuous state problems. With NFQ a powerful neural RL method for continuous domains is described. However, it has a number of issues that are discussed. The thesis then continues to propose ensembles to mitigate those problems and make NFQ more reliable. A number of ways to use ensembles in an NFQ context are discussed and evaluated on a number of benchmark problems. Finally, the thesis gives some explanations why ensembles work here and outlines ways to adapt the methods to continuous actions.
- In **Chapter 6** the methods for self-assessment using uncertainty awareness are to be adapted to continuous domains. It is argued that this is not possible, since in the continuous domain setting no distribution over MDPs is available. Instead, one can use fitted policy evaluation (FPE), which can be improved by alternating the function approximator and/or dataset compared to those used for policy generation. Experiments show that it is possible to use the quality measure derived by FPE to select good and reject poor policies from an ensemble or derive weighted ensembles.
- **Chapter 7** finally combines previous ideas. First, a scenario is discussed where one has a number of policies for different parameterizations of the same problem. The current parameterization is unknown. The task is then to select policies suitable for this parameterization. The main part of this chapter details the evolving ensemble approach. The idea is to use an

ensemble that is regularly updated to be able to deal with a slowly changing environment. The updates are performed by adding new policies, generated from recent observations, and remove old, supposedly obsolete policies.

- **Chapter 8** summarizes the thesis and outlines possible areas for future research.

## 1.6 Publications

Parts of this thesis have already been published as peer-reviewed papers at international conferences:

- A. Hans and S. Udluft (2009). “Efficient Uncertainty Propagation for Reinforcement Learning with Limited Data”. In: *Proceedings of the International Conference on Artificial Neural Networks*.

This paper introduces the diagonal approximation of uncertainty-incorporating policy iteration (DUIPI) as an efficient approximation of full-matrix uncertainty propagation. Its contents are covered by Chapters 3 and 4.

- A. Hans and S. Udluft (2010b). “Uncertainty Propagation for Efficient Exploration in Reinforcement Learning”. In: *Proceedings of the 19th European Conference on Artificial Intelligence*. IOS Press.

Here it is shown how the knowledge of uncertainty can be used to explore efficiently. Further, a variant of DUIPI called diagonal approximation of uncertainty-incorporating policy iteration with  $Q$  modification (DUIPI-QM) is presented. The section about exploration in Chapter 4 uses contents of this paper.

- A. Hans, S. Duell, and S. Udluft (2011). “Agent Self-Assessment: Determining Policy Quality Without Execution”. In: *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*.

Finally, this paper argues that looking only at the  $Q$ -function when assessing the quality of a policy can be misleading, which can be corrected by considering the uncertainty as well. The contents of this paper are contained within the self-assessment section of Chapter 4.

- A. Hans and S. Udluft (2010a). “Ensembles of Neural Networks for Robust Reinforcement Learning”. In: *Proceedings of the 9th IEEE International Conference on Machine Learning and Applications*. IEEE, pp. 401–406.

This is the first paper to suggest and evaluate ensembles in an NFQ context. It served as the basis for parts of Chapter 5.

- A. Hans and S. Udluft (2011). “Ensemble Usage for More Reliable Policy Identification in Reinforcement Learning”. In: *Proceedings of the 19th European Symposium on Artificial Neural Networks*.

Here, more empirical evidence is presented that ensembles improve the reliability of NFQ. Further, it is proposed to build ensembles from successive NFQ iterations as a computationally cheap alternative.

Chapters 6 and 7 contain completely new, previously unpublished material.

The following (co-authored) papers are not covered by this thesis, but their contents are related:

- A. Hans, D. Schneegaß, A. M. Schäfer, and S. Udluft (2008). “Safe Exploration for Reinforcement Learning”. In: *Proceedings of the 16th European Symposium on Artificial Neural Networks*, pp. 413–418.
- S. Duell, A. Hans, and S. Udluft (2010). “The Markov Decision Process Extraction Network”. In: *Proceedings of the 18th European Symposium on Artificial Neural Networks*.
- S. Duell, L. Weichbrodt, A. Hans, and S. Udluft (2012). “Recurrent Neural State Estimation in Domains with Long-Term Dependencies”. In: *Proceedings of the 20th European Symposium on Artificial Neural Networks*.



# 2

## Reinforcement Learning

This chapter gives an overview of reinforcement learning (RL) and introduces methods that later parts of this thesis will build upon.

### 2.1 Overview

RL denotes a field of machine learning that is concerned with problems of sequential decision making. Often the notion of an agent acting in some environment is used. When dealing with time-discrete systems, to which we limit the discussion here, at each time step  $t = 1, 2, \dots, N$  the agent observes the environment's current state  $s \in \mathcal{S}$  and can carry out some action  $a \in \mathcal{A}$ , where  $\mathcal{S}$  is the state space and  $\mathcal{A}$  the action space, respectively. While transiting, at each time step  $t$  the agent receives a reward  $r_t \in \mathbb{R}$ . The agent acts according to a policy  $\pi$  determining the (probability for an) action to carry out based on the current state. The policy can either be deterministic, i.e.,  $\pi : \mathcal{S} \mapsto \mathcal{A}$ , or stochastic, i.e.,  $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ .<sup>1</sup> Usually, the aim in RL is to find a policy that, when followed, optimizes the expected sum of future rewards (the return)

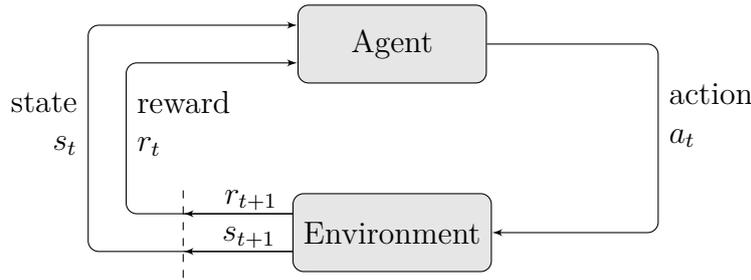
$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots, \quad (2.1)$$

which can be discounted by setting the discount factor  $\gamma \in [0, 1]$  to a value smaller than 1. Discounting is necessary if rewards in the near future are to be weighted higher than those further away. Another, more technical reason for discounting is keeping the return from going to infinity, which is necessary for methods based on value functions (discussed later in this chapter). In general, we choose large discount factors like  $\gamma = 0.95$  or  $\gamma = 0.975$ , because we are interested in optimizing the long-term reward.

The most interesting property of RL is that it learns from *observations*. In the classical setting, the RL agent interacts with the environment and learns from this interaction which action is optimal in a given state. In an alternative setting,

---

<sup>1</sup>Obviously, a deterministic policy can be expressed as a stochastic one by using discrete probabilities of 0 and 1.



**Figure 2.1:** Interaction between agent and environment. At time step  $t$  the agent observes state  $s_t$  and reward  $r_t$  from the environment. Based on this information it influences the environment using action  $a_t$ , which causes a transition to state  $s_{t+1}$ .

an RL method is given a set of observations and must derive a policy from this set without additional interaction. In both cases, an observation can be considered as a tuple  $(s, a, s', r)$ , meaning that in state  $s$  action  $a$  was executed, causing a transition to state  $s'$  and a reward  $r$ .

## 2.2 Markov Decision Processes

An RL problem is commonly described as a Markov decision process (MDP)

$$M := (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}), \quad (2.2)$$

composed of:

- A state space  $\mathcal{S}$ .
- An action space  $\mathcal{A}$ . A definition as  $\mathcal{A}(s)$  is common as well if the set of available actions is dependent on the state  $s$ .
- A transition function  $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ , giving the probability of reaching a successor state  $s'$  from a state-action pair  $(s, a)$ .
- A reward function  $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ , which gives the expected reward of a transition.

A defining feature of an MDP is the *Markov property*. If it is fulfilled, the probability of a successor state  $s_{t+1}$  only depends on the state-action pair  $(s_t, a_t)$ , all previous states and actions are irrelevant, i.e.,

$$\Pr(s_{t+1}|s_t, a_t) = \Pr(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0). \quad (2.3)$$

Likewise, the reward expectation is only dependent on the current transition  $s_t \rightarrow_{a_t} s_{t+1}$ , i.e.,

$$\mathbf{E} \{r_{t+1} | s_{t+1}, a_t, s_t\} = \mathbf{E} \{r_{t+1} | s_{t+1}, a_t, s_t, a_{t-1}, s_{t-1}, \dots, s_1, a_0, s_0\} \quad (2.4)$$

Consider for example an object moving along a straight line with some velocity as an MDP. If the state contains the current position only, the Markov property is not fulfilled, since it is not possible to derive the successor state, i.e., the position in the next time step, from the current one. Additionally, knowledge of the object's velocity is required. In this case, we deal with a higher-order MDP (order of 2 in this case), because we can derive the velocity from two consecutive states  $(s_{t-1}, s_t)$  and this way determine  $s_{t+1}$ . In other words, by defining a new state space  $\tilde{\mathcal{S}} := \{\tilde{s}_t | \tilde{s}_t = (s_{t-1}, s_t)\}$  it is possible to restore the Markov property. An equivalent solution would directly include the velocity into the state.

Many RL methods depend on the Markov property. If it is violated, those methods will in general not work. However, if the violation is not severe, the application of methods assuming the state to be Markovian is often uncritical (Sutton and Barto, 1998, p. 64).

The framework dealing explicitly with non-Markovian environments is that of the partially observable Markov decision process (POMDP) (Kaelbling, Littman, and Moore, 1996; Kaelbling, Littman, and Cassandra, 1998). A POMDP  $M' := (\mathcal{S}, \mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{O})$  contains in addition to the components of an MDP an observation space  $\mathcal{Z}$  and an observation function  $\mathcal{O} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{Z}$ . In a POMDP the true state  $s \in \mathcal{S}$  is hidden and only observations  $z \in \mathcal{Z}$  are available. From a series of observations a *belief state* as a distribution over possible hidden states is constructed. The problem then is to find a policy that optimally maps belief states to actions.

A different approach is the use of a *state estimator* that takes a series of  $T$  observations  $z_{t-T}, z_{t-T+1}, \dots, z_t$  and reconstructs the current Markovian state  $s_t$  from them (e.g., Schäfer and Udluft, 2005; Duell, Hans, and Udluft, 2010). This reconstructed Markovian state can then be used with approaches that expect the state to be Markovian.

In this thesis the environment is assumed to be an MDP with a Markovian state representation.

## 2.3 Value Functions

For every MDP so called *value functions* can be defined that give the expected return when starting from a specific state ( $V$ -function) or a state-action pair ( $Q$ -function). In any case, they assume some fixed policy. Note that while there exists exactly one value function for a specific policy, multiple policies can yield the same value function.

The value function  $V^\pi : \mathcal{S} \mapsto \mathbb{R}$  gives the expected return for policy  $\pi$  and a state  $s$ , i.e, the expected sum of discounted future rewards when starting from  $s$  and strictly following  $\pi$ :

$$V^\pi(s) = \mathbf{E}_\pi \{R_t | s_t = s\} = \mathbf{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | s_t = s \right\}. \quad (2.5)$$

$\mathbf{E}_\pi$  here denotes the expected value under the assumption that policy  $\pi$  is followed.  $V^\pi$  is known as the *state-value function* for policy  $\pi$ . It can be formulated in a recursive way:

$$\begin{aligned} V^\pi(s) &= \mathbf{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | s_t = s \right\} \\ &= \mathbf{E}_\pi \left\{ r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s \right\} \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, \pi(s)) \left[ \mathcal{R}(s, \pi(s), s') + \gamma \mathbf{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s' \right\} \right] \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, \pi(s)) [\mathcal{R}(s, \pi(s), s') + \gamma V^\pi(s')]. \end{aligned} \quad (2.6)$$

Equations (2.5) and (2.6) assume a deterministic policy  $\pi$ . For a stochastic policy  $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ , which gives the probability of executing action  $a$  in state  $s$ , we have to take another expectation:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) [\mathcal{R}(s, a, s') + \gamma V^\pi(s')]. \quad (2.7)$$

The same can be formulated for state-action pairs, leading to the  $Q$ -function, which gives the expected return for a state-action pair  $(s, a)$  when executing  $a$  in  $s$  and afterwards strictly following policy  $\pi$ :

$$Q^\pi(s, a) = \mathbf{E}_\pi \{R_t | s_t = s, a_t = a\} = \mathbf{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | s_t = s, a_t = a \right\}. \quad (2.8)$$

$Q^\pi$  denotes the *action-value function* of  $\pi$ . Analog to the recursive formulation of  $V^\pi$ , a recursive formulation of  $Q^\pi$  is possible:

$$\begin{aligned} Q^\pi(s, a) &= \mathbf{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} | s_t = s, a_t = a \right\} \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) [\mathcal{R}(s, \pi(s), s') + \gamma V^\pi(s')]. \end{aligned} \quad (2.9)$$

Note that

$$V^\pi(s) = Q^\pi(s, \pi(s)).$$

Again, equations (2.8) and (2.9) assume a deterministic policy. For a stochastic

policy we have

$$\begin{aligned} Q^\pi(s, a) &= \mathbf{E}_\pi \left\{ \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s, a_t = a \right\} \\ &= \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) [\mathcal{R}(s, \pi(s), s') + \gamma V^\pi(s')]. \end{aligned} \quad (2.10)$$

The equations defining  $V^\pi$  and  $Q^\pi$  are called *Bellman equations*.

The goal of RL lies in finding a policy that achieves as much reward as possible in the long run, i.e, that maximizes the return. Value functions define a partial ordering over policies. A policy  $\pi$  is better than or equal to a policy  $\pi'$  if and only if its value function gives an equal or greater return for all states. More formally,

$$\pi \geq \pi' \Leftrightarrow \forall s \in \mathcal{S} : V^\pi(s) \geq V^{\pi'}(s). \quad (2.11)$$

For any given MDP there is at least one policy whose value function is greater than or equal to all other value functions. Such a policy is known as an *optimal policy*. While there may be more than one optimal policy, all of them share the same state-value function, called the *optimal state-value function*,

$$V^*(s) = \max_{\pi} V^\pi(s). \quad (2.12)$$

Likewise, they share the same *optimal action-value function*,

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a). \quad (2.13)$$

Since  $V^*$  is the state-value function of an optimal policy, it can be expressed in terms of the action-value function without referring to a specific policy as

$$V^*(s) = \max_{a \in \mathcal{A}} Q^{\pi^*}(s, a), \quad (2.14)$$

because an optimal policy takes the action that maximizes the return or equivalently maximizes the  $Q$ -value for a given state. From this, using equation (2.9), we obtain

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) [\mathcal{R}(s, \pi(s), s') + \gamma V^*(s')], \quad (2.15)$$

which is known as the *Bellman optimality equation for  $V^*$* . The same can be expressed in terms of state-action pairs as

$$\begin{aligned} Q^*(s, a) &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) [\mathcal{R}(s, \pi(s), s') + \gamma V^*(s')], \\ &= \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) \left[ \mathcal{R}(s, \pi(s), s') + \gamma \max_{a'} Q^*(s', a') \right], \end{aligned} \quad (2.16)$$

which is known as the *Bellman optimality equation for  $Q^*$* .

Since  $Q^* = Q^{\pi^*}$  is the action-value function of the optimal policy  $\pi^*$  and the optimal policy always chooses the action maximizing the  $Q$ -function for the given state, the optimal policy can be expressed in terms of  $Q^*$ :

$$\pi^*(s) := \arg \max_{a \in \mathcal{A}} Q^*(s, a). \quad (2.17)$$

Obviously, knowledge of  $Q^*$  is sufficient to know the optimal policy  $\pi^*$ . In fact, many RL methods build on action-value functions. Instead of directly looking for the optimal policy, they try to determine its value function, as from the optimal value function the optimal policy follows trivially.

## 2.4 Dynamic Programming

Methods belonging to the class of dynamic programming (DP) can be used to determine the value function for a given policy or the optimal value function when the MDP is completely known, in particular the state transition function  $\mathcal{P}$  and the reward function  $\mathcal{R}$ .

While the term *dynamic programming* was introduced by Bellman (1957) for methods that solve optimal control problems (exactly the methods discussed in this section), today it is used to identify a broad class of algorithms. The common characteristic of those algorithms is that they solve a problem by dividing it into smaller and smaller sub-problems until the sub-problems are small enough to be solved directly. Then the solutions to the smallest sub-problems are put together to solutions of larger sub-problems, until eventually the complete problem is solved. DP is similar to classic divide-and-conquer algorithms (for example, sorting algorithms like quick sort and merge sort), but in addition DP stores intermediate results (solutions to sub-problems) and re-uses them multiple times, whereas classic divide-and-conquer algorithms use intermediate results only once (Cormen, Leiserson, Rivest, and Stein, 2009). DP can be used when the Bellman optimality principle holds, which states that any part of an optimal plan must be in itself optimal. In the case of policies this means “that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision” (Bellman, 1957, p. 83). Aside from RL, algorithms that use DP include Dijkstra’s algorithm for the shortest path problem (Dijkstra, 1959) and many string algorithms, for instance for calculating the longest common subsequence (Cormen, Leiserson, Rivest, and Stein, 2009) or the Levenshtein distance of two strings, i.e., the number of operations necessary to transform one string into the other (Levenshtein, 1966).

All DP methods in RL are based on Bellman equations. In general, they turn a

Bellman equation into an update rule. Starting from an arbitrary initial guess repeatedly applying the update rule leads to a better and better estimate, in the limit the estimate tends to the true value function.

### 2.4.1 Policy Evaluation, Improvement, and Iteration

Policy evaluation can be used to determine the action-value function of a policy  $\pi$ . Starting with an arbitrary initial guess  $Q_0^\pi$ , the following update rule (based on the Bellman equation for  $V^\pi$ , equation (2.9)) is applied repeatedly for all state-action pairs:

$$Q_{i+1}^\pi(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) [\mathcal{R}(s, a, s') + \gamma Q_i^\pi(s', \pi(s'))]. \quad (2.18)$$

With  $i \rightarrow \infty$ ,  $Q_i^\pi$  converges to the true value function  $Q^\pi$ . In practice, one stops when the change from one iteration to the next falls below a pre-defined threshold  $\varepsilon$  (Algorithm 1).

---

#### Algorithm 1: Policy Evaluation

---

**Input:** policy  $\pi$ , transition probabilities  $\mathcal{P}$ , reward function  $\mathcal{R}$ , discount factor  $\gamma$ , error threshold  $\varepsilon$

**Result:** estimate  $\hat{Q}^\pi$  with  $\max_{s,a} |Q^\pi(s, a) - \hat{Q}^\pi(s, a)| < \varepsilon$

```

1 begin
   | Initial guess as zero
2   |  $Q_0^\pi \leftarrow 0$ 
3   |  $e \leftarrow \infty$ 
4   | while  $e \geq \varepsilon$  do
5   |   |  $e \leftarrow 0$ 
6   |   | for  $\forall (s, a) \in \mathcal{S} \times \mathcal{A}$  do
7   |   |   |  $Q_{i+1}^\pi(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) [\mathcal{R}(s, a, s') + \gamma Q_i^\pi(s', \pi(s'))]$ 
8   |   |   | Record maximum update
9   |   |   | if  $|Q_{i+1}^\pi(s, a) - Q_i^\pi(s, a)| > e$  then
10  |   |   |   |  $e \leftarrow |Q_{i+1}^\pi(s, a) - Q_i^\pi(s, a)|$ 
11  |   |  $i \leftarrow i + 1$ 
   | return  $Q_i^\pi$ 

```

---

Once the action-value function  $Q^\pi$  is known, it can be used to improve the policy in a step called *policy improvement*. Policy improvement updates a policy to select the action maximizing  $Q^\pi$  in each state, i.e.,

$$\pi'(s) \leftarrow \arg \max_{a \in \mathcal{A}} Q^\pi(s, a) \quad \forall s \in \mathcal{S} \quad (2.19)$$

Policy evaluation and policy improvement can be combined to policy iteration:

Starting with an arbitrary policy  $\pi_0$ , this policy is evaluated resulting in  $Q^{\pi_0}$ , which is then used to improve the policy. The result of the improvement step is stored as  $\pi_1$ , which again is evaluated and improved, etc. Since every update is a strict improvement over the previous policy, unless the previous policy is already optimal, and for a finite MDP there is only a finite number of policies, policy iteration must converge to an optimal policy in a finite number of iterations (Sutton and Barto, 1998).

## 2.4.2 Value Iteration

A major drawback of policy iteration is that in each iteration it requires a complete policy evaluation step, which itself consists of a number of iterations over the complete state set. Luckily, it is possible to combine policy evaluation and policy improvement into one step, where policy improvement is done immediately after the update of the value function without waiting for it to converge. The resulting algorithm is called *value iteration* and has the same convergence properties as policy iteration (Bertsekas, 1987). It can be seen as turning the Bellman optimality equation (2.16) into an update rule, namely

$$Q_{i+1}^*(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) \left[ \mathcal{R}(s, a, s') + \max_{a'} Q_i^*(s', a') \right], \quad (2.20)$$

that is applied to each state-action pair  $(s, a) \in \mathcal{S} \times \mathcal{A}$ . Here the policy is implicitly assumed as  $\pi_i(s) := \arg \max_{a \in \mathcal{A}} Q_i^*(s, a)$ . Algorithm 2 summarizes value iteration.

## 2.4.3 Learning from Observations

The methods discussed so far assume complete knowledge of the MDP. In contrast, RL means learning from interaction with the environment or, more generally, from observations in the form of tuples  $(s, a, s', r)$ , denoting a transition from a state  $s$  with action  $a$  to a successor state  $s'$  with reward  $r$ . Although methods based on DP require complete knowledge of the MDP, it is possible to use them in an RL setting, because the MDP's unknown components can be estimated from observation tuples. Namely, those are the transition probabilities  $\mathcal{P}$  and the reward function  $\mathcal{R}$ .

Straightforwardly,  $\mathcal{P}$  can be estimated by means of relative frequency. Let  $n_{s,a}$  denote the number of occurrences of state-action pair  $(s, a)$  in the set of observations, i.e., the number of transitions starting from  $s$  with action  $a$ . Let further  $n_{s,a,s'}$  denote the number of occurrences of the transition  $s \rightarrow_a s'$ . The probability

**Algorithm 2:** Value Iteration

**Input:** transition probabilities  $\mathcal{P}$ , reward function  $\mathcal{R}$ , discount factor  $\gamma$ , error threshold  $\varepsilon$

**Result:** estimate  $\hat{Q}^*$  with  $\max_{s,a} |Q^*(s,a) - \hat{Q}^*(s,a)| < \varepsilon$

```

1 begin
  Initial guess as zero
2    $Q_0^* \leftarrow 0$ 
3    $e \leftarrow \infty$ 
4   while  $e \geq \varepsilon$  do
5      $e \leftarrow 0$ 
6     for  $\forall (s,a) \in \mathcal{S} \times \mathcal{A}$  do
7        $Q_{i+1}^*(s,a) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) [\mathcal{R}(s,a,s') + \gamma \max_{a'} Q_i^*(s',a')]$ 
       Record maximum update
8       if  $|Q_{i+1}^*(s,a) - Q_i^*(s,a)| > e$  then
9          $e \leftarrow |Q_{i+1}^*(s,a) - Q_i^*(s,a)|$ 
10     $i \leftarrow i + 1$ 
11  return  $Q_i^*$ 

```

of a transition to  $s'$  from  $s$  with  $a$  can then be estimated as

$$\hat{\mathcal{P}}(s'|s,a) := \frac{n_{s,a}}{n_{s,a,s'}}. \quad (2.21)$$

For a specific state-action pair  $(s,a)$ ,  $\hat{\mathcal{P}}(s'|s,a)$  defines a probability distribution over all possible successor states  $s'$ , therefore  $\sum_{s' \in \mathcal{S}} \hat{\mathcal{P}}(s'|s,a) = 1$ . Also,  $n_{s,a} = \sum_{s' \in \mathcal{S}} n_{s,a,s'}$ .

To estimate the reward function  $\mathcal{R}$ , the mean can be used. Let  $r_{s,a,s'}^i, i = 1, \dots, n_{s,a,s'}$  denote the list of rewards that occurred with the transition  $s \rightarrow_a s'$ . Then  $\mathcal{R}$  can be estimated as

$$\hat{\mathcal{R}}(s,a,s') := \frac{\sum_{i=1}^{n_{s,a,s'}} r_{s,a,s'}^i}{n_{s,a,s'}}. \quad (2.22)$$

Obviously, the more observations available, the better the above estimates are. In practice, they often work well even for small numbers of observations. However, if the estimates are flawed, the impact on the resulting policy can be dramatic, since in the DP methods the estimates are taken for the true value again and again. This problem will be discussed in more detail in the next chapter. A solution lies in not only using the estimates  $\hat{\mathcal{P}}$  and  $\hat{\mathcal{R}}$ , but also considering their uncertainties.

## 2.5 Temporal-Difference Learning

Temporal-difference (TD) methods (Sutton, 1988) learn a value function directly from experience, i.e., interaction with the environment. Whenever a transition occurs, they use the corresponding observation tuple to update the current value function estimate. More precisely, TD methods determine the so-called *TD error* and update the value function to decrease that error. Recall the Bellman equation for the state-value function  $V^\pi$ ,

$$V^\pi(s) = \mathbf{E}_\pi \left\{ r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s \right\}.$$

If  $V^\pi$  is correct, for an observation  $(s, a, s', r)$  on average  $V^\pi(s) = r + \gamma V^\pi(s')$  should hold. From this, we can derive the TD error as the difference of the two terms left and right of the equation sign:

$$\text{TDE} := r + \gamma V^\pi(s') - V^\pi(s). \quad (2.23)$$

TD learning for  $V^\pi$  starts with an arbitrary guess for  $V^\pi$ , e.g.,  $\hat{V}^\pi := 0$ , and then improves it with each observation using an update rule based on the TD error. For an observation  $(s, a, s', r)$ ,  $\hat{V}^\pi(s)$  is updated according to

$$\hat{V}^\pi(s) \leftarrow \hat{V}^\pi(s) + \alpha \left[ r + \gamma \hat{V}^\pi(s') - \hat{V}^\pi(s) \right], \quad (2.24)$$

where  $\alpha \in (0, 1]$  is a learning rate. A small learning rate is necessary to estimate values in stochastic environments.  $V^\pi$  gives the *expected* return, but the observation tuple used to improve the estimate is just *one* observation contributing to the expectation. With a learning rate  $\alpha = 1$  the estimated value would jump from one value to another with different observations, instead of converging to the expected value. If the environment is known to be deterministic (both, state transitions and rewards), a learning rate  $\alpha = 1$  may be used.

To determine an optimal policy with TD methods, there are two common algorithms: SARSA and  $Q$ -learning.

SARSA (Rummery and Niranjan, 1994) estimates the  $Q$ -function of the currently followed policy. It is therefore an *on-policy* method. With an observation tuple  $(s, a, r, s', a')$  (in addition to previously used observation tuples, the action selected in the successor state is included as well) the following update rule is used to update the estimate:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma Q(s', a') - Q(s, a) \right]. \quad (2.25)$$

Similar to the update of  $V^\pi$ , equation (2.24), the current observation is used to determine the error of the current estimate. The estimate is then corrected by the  $\alpha$  weighted error.

If the followed policy chooses actions by maximizing the current  $Q$  estimate, it is

automatically improved. This can be seen as something like a policy improvement step, while the  $Q$  update corresponds to the policy evaluation step. Furthermore, it should be ensured that every now and then alternative actions are tried. For example, this can be achieved by using  $\varepsilon$ -greedy exploration, where with probability  $\varepsilon$  a random action is chosen and with probability  $1 - \varepsilon$  the action maximizing the current  $Q$  estimate. SARSA is an on-policy method because it only uses observations generated by the current policy. In particular, it uses  $Q(s', a')$  which depends on the successor state and the action chosen by  $\pi$  in this state.

$Q$ -learning (Watkins, 1989) is a TD method that always estimates the optimal action-value function  $Q^*$ , regardless of the followed policy. It does this by being an *off-policy* method that uses an update rule based on the Bellman optimality equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[ r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s, a) \right]. \quad (2.26)$$

While SARSA uses the  $Q$ -value of the actually chosen action in the successor state,  $Q$ -learning assumes the action maximizing the current  $Q$ -value estimate, i.e., the action currently considered best. Given that each state-action pair  $(s, a)$  is visited infinitely often,  $Q$ -learning's  $Q$ -function estimate converges to the optimal action-value function  $Q^*$ .

## 2.6 Data-Efficiency

When evaluating RL methods, *data-efficiency* is an important criterion. A method that needs less observations from interaction with the environment to achieve a certain policy quality is considered more data-efficient than a method requiring more observations. For many applications data-efficiency is crucial, as exploration of the actual system is often expensive and time-consuming.

The methods discussed so far fall into two categories: model-free and model-based methods. While TD methods learn directly from observations, DP methods use a model of the environment. Furthermore, they can be categorized into on-line and off-line (or batch-mode) methods. TD methods learn directly (on-line) while interacting with the environment. In contrast, when learning from observations DP methods must first estimate a model of the environment and use this model to determine a policy. Both, the model estimation as well as the policy determination, are done off-line and usually from a set (or batch) of observations. In general, model-based methods are more data-efficient, since they effectively re-use observations (Atkeson and Santamaria, 1997). Classic TD methods use every observation only once; an DP approach with an estimated model re-uses the observations by means of the model.

An approach to increase the data-efficiency of TD methods is *experience replay* (Lin, 1992). Observations are stored and used multiple times to update the action-

value estimate  $\hat{Q}$ , thereby using TD learning in a batch-mode setting. Another approach to increase the data-efficiency of TD methods are so-called *eligibility traces*, where for each state or state-action pair an additional value, the eligibility  $e_t$ , is stored. Whenever a TD update occurs, not only the value estimate of the current state or state-action pair is modified, but also all others according to their value  $e_t(s)$  or  $e_t(s, a)$ , respectively. Whenever a state or state-action pair is visited, its eligibility value is increased. In each time step, all eligibility values decay. As a result, recently visited states or state-action pairs have a higher eligibility value than seldom visited ones. The decay is influenced by a parameter  $\lambda$ , with  $\lambda = 0$  eligibility values of all states or state-action pairs except the most recently visited one are set to zero, thus with  $\lambda = 0$  the method is equal to those not using eligibility traces at all. General TD learning with eligibility traces is called TD( $\lambda$ ). Rummery and Niranjan (1994) proposed SARSA( $\lambda$ ), different versions of  $Q$ -learning with eligibility traces, i.e., Q( $\lambda$ ), have been proposed by Watkins (1989) and Peng and Williams (1996).

Although model-based DP approaches are very data-efficient, they have two major drawbacks. First, they are very sensitive w.r.t. modeling errors. They determine an optimal policy for the estimated MDP. Even if the estimate is close to the real MDP, a single mis-estimated transition probability can have a dramatic impact on the resulting policy. Despite being optimal for the estimated MDP, it might perform insufficiently when applied to the real system. In Chapter 3 this problem is discussed in more detail. Another problem is the computational efficiency when used on-line. For the best possible performance, each new observation should be used as soon as possible. For DP this means that after each interaction with the environment the estimates of  $\mathcal{P}$  and  $\mathcal{R}$  have to be recalculated, afterwards a complete run of value iteration has to be done. All in all a lot of calculations are necessary, although the state-value function after the new observation is most probably not very different from the previous. Exactly this fact, that one new observation on average has only little influence on the resulting state-value function, is used by *prioritized sweeping* (Moore and Atkeson, 1993). It works similar to value iteration but updates only values for state-action pairs whose estimated value would change significantly. To this end the algorithm keeps a queue of these state-action pairs, prioritized by the size of their changes. When the state-action pair from the top of the queue is updated, the effect on each of its predecessor pairs is calculated. If the value change of a predecessor pair would be larger than some small threshold, it is inserted into the queue. Once the value of a state-action pair is updated, it is removed from the queue. This is repeated until the queue is empty or some maximum number of state-action pairs has been processed.

## 2.7 Function Approximation

All methods discussed so far work only for reasonably small discrete state and action spaces, because they assume that the value of each state-action pair is an entry of a table in memory. For MDPs with a large discrete or continuous state space this becomes impractical. First, there is the storage problem—for each possible state-action pair its value must be stored, which becomes impossible for such MDPs. Second, even if we were able to store all those values, we would face the problem of data-efficiency—it would take a lot of time and data (observations) to fill the entries of the table with accurate values. Both issues are addressed by *generalization*. Instead of storing each value independently, some representation is used. This representation could be a function approximator parameterized by a vector  $\vec{\theta}$ . The number of parameters in  $\vec{\theta}$  is usually much smaller than the number of state-action pairs. Learning in this context means updating  $\vec{\theta}$ . This way an update due to an observation involving only a single specific state-action pair potentially changes the values of many more state-action pairs—the update is *generalized*.

A very simple form of generalization is *state aggregation*, where instead of storing the value of each single state a number of states are aggregated and represented by a single value. When dealing with a continuous state space, this can be done by discretizing each dimension of the state space, resulting in hypercubes. All states within one hypercube are then aggregated to a single discrete state. State aggregation is relatively simple to implement and, given that the discretization is done with care, can lead to acceptable results. In general, however, it leads to a violation of the Markov property, since it is unknown where exactly in the hypercube a state is located, but the probability distribution of successor states depends on the exact location. Therefore, the quality of the resulting policy is highly problem-dependent.

A better approach is using a “proper” function approximator to represent the value function. Let  $h_{\vec{\theta}}: \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$  denote a smooth and differentiable function that is parameterized by  $\vec{\theta}$ . We can then use *gradient descent* (Widrow and Hoff, 1960; Bishop, 1995) to change  $\vec{\theta}$  in a way that the difference between  $h_{\vec{\theta}}$  and the true function becomes smaller. For gradient descent, an error function giving the current deviation between desired and actual output of the function approximator must be defined. Then the error function is differentiated w.r.t. to  $\vec{\theta}$ . The resulting vector of partial derivatives is called the *gradient* of the error function. It can then be used to change  $\vec{\theta}$  in the direction of the negative gradient to decrease  $h_{\vec{\theta}}$ 's error. To use function approximation with TD methods, the TD error is used as error function (Sutton, 1988). Where a table-based TD approach updates directly the table entries, TD learning with function approximation uses the TD error to update the parameter vector  $\vec{\theta}$ . Popular function approximators used for TD learning are artificial neural networks. For example, Tesauro's Td-Gammon used  $Q$ -learning with neural networks (Tesauro, 1994). Also linear function ap-

proximators like the “cerebellar model articulator controller” (CMAC) (Albus, 1971), which in RL is known as tile coding (Watkins, 1989), have been used (e.g., Watkins, 1989; Sutton, 1996). In tile coding, the state space is discretized by tilings (grids). Multiple tilings of different coarseness and with different offsets are used. Every tile of a tiling is a receptive field for one binary feature. If a state or state-action pair lies within a tile, the corresponding binary feature equals 1 (“on”). To make a prediction, the values of all features that are “on” are summed. In tile coding, the vector of all feature values corresponds to the parameter vector  $\vec{\theta}$ . To learn, i.e., update  $\vec{\theta}$ , the values for the active features are adjusted in the direction of decreasing error.

A class of methods important for this thesis is *fitted value iteration* (Gordon, 1995). It can be considered as a form of approximate dynamic programming (ADP), since it does iterations just like DP based value iteration. Instead of storing the value function tabularly, a function approximator is used. In each iteration  $i$ , new values  $V_i$  of a set of sample states  $X$  are calculated. Then a function approximator  $h_i$  is trained to map  $X$  to  $V_i$ . In the next iteration,  $i + 1$ ,  $h_i$  takes the place of the value function in the Bellman update equation to calculate  $V_{i+1}$ . Ormonoit and Sen (2002) used this framework with kernel-based approximators to learn a  $Q$ -function, Lagoudakis and Parr (2003) used a linear combination of basis functions. Ernst, Geurts, and Wehenkel (2003) were the first to use observations of the environment  $((s, a, s', r)$  tuples) instead of model-generated samples. They later called the method fitted  $Q$ -iteration (FQI) (Ernst, Geurts, and Wehenkel, 2005). Riedmiller (2005) combined FQI with neural networks, resulting in neural fitted  $Q$ -iteration (NFQ) .

A remarkable feature of FQI is that it can be combined with any function approximator, while earlier approaches like Sutton (1988) or Tsitsiklis (1994) depend on parametric methods involving a parameter vector  $\vec{\theta}$  (Ernst, Geurts, and Wehenkel, 2005). In Chapters 5 and 6 FQI with tree-based regression methods and neural networks is employed and described it in more detail.

Another notable approach to approximating the optimal action-value function  $Q^*$  is *rewards regression* (Schneegaß, Udluft, and Martinetz, 2006). The basic idea of rewards regression is to reformulate the regression task so that the reward observations are used as targets. While the kernel rewards regression approach (Schneegaß, Udluft, and Martinetz, 2006) still needs to iterate, it was later extended to explicit kernel rewards regression, where the problem is solved in one step by means of quadratic programming (Schneegaß, Udluft, and Martinetz, 2007b). Similarly, the neural rewards regression approach is a reformulation of the task of learning  $Q^*$  into a neural topology that has the rewards as targets (Schneegaß, Udluft, and Martinetz, 2007a).

## 2.8 Policy Gradient and Policy Search Methods

Besides methods approximating the optimal action-value function (with the aim of deriving a near-optimal policy), there are two groups of methods that directly search the policy space.

The first group are *policy gradient* methods. They are called so because they use the gradient of the policy function w.r.t. its parameters. In order to do so, the policy must be differentiable. If the policy is represented by a neural network, this is normally the case. Since a deterministic discrete-action policy function is not differentiable, in the case of discrete actions a stochastic policy must be used. On the other hand, such an approach can deal directly with continuous actions without the need to discretize whatsoever. Then, using the gradient w.r.t. policy parameters with the policy plugged into some optimality function, gradient ascent is performed to improve the policy (the parameters of the policy are changed so that the optimality function is maximized). Hafner and Riedmiller (2011) introduced *neural fitted Q-iteration with continuous actions* (NFQCA), which adapts the NFQ idea to the continuous actions setting. While NFQ resembles value iteration, NFQCA corresponds to policy iteration. A policy network is added that maps a state to a continuous action. The output of this network is fed into the  $Q$  network already present in NFQ. One iteration then proceeds in two steps: first, using both networks, new  $Q$  targets are calculated, and then a new  $Q$  network is learned (policy evaluation). Second, the  $Q$  network is frozen and the policy improved by doing gradient ascent on the policy network (policy improvement). The *policy gradient neural rewards regression* approach (PGNRR) due to Schneegaß, Udluft, and Martinetz (2007c) integrates both steps into one network topology by means of shared weights and selective gradient flow control. PGNRR has no need for iteration, the policy is learned by one network training step.

While NFQCA and PGNRR use the value function as optimality function, there are approaches depending on policy roll-outs. They determine the quality of a policy by actually running it. From the performances of differently parameterized policies the gradient can be determined and the policy improved in positive gradient direction. The success of value function based approach depends highly on the Markov property. With approaches building on actual roll-outs, the dependency is less strong. As a consequence, the methods are more robust w.r.t. a violation of the Markov property. Naturally, they are only applicable when data-efficiency is not required, for instance because a simulator is available. For examples for such methods see Riedmiller, Peters, and Schaal (2007), Sehnke et al. (2010).

The recurrent control neural network (RCNN) (Schäfer, Udluft, and Zimmermann, 2007) is a method that lies between value-function based approaches and approaches depending on actual roll-outs. It is a neural network composed of two parts: a recurrent sub-network modeling the environment (environment model) and a control network (MLP mapping current (internal) state of the environment

model to an action). The training process proceeds in two steps: first, observations of the environment are used to learn the environment model. Second, this network is frozen and used to predict the  $n$ -step return of the policy currently encoded in the control network, where  $n$  gives the future horizon and is a parameter of the topology. Like in other policy gradient methods, gradient ascent is performed. In the RCNN approach, the weights of the control network are adjusted to maximize the return, i.e., the sum of rewards from each future time-step. Because of the explicit summation using the future time-steps, the RCNN can be considered as a “virtual Monte-Carlo policy gradient” method (Schäfer, 2008).

The second group contains methods often called *policy search* or *direct policy search* methods. They are similar to policy gradient methods in that they also search the policy space directly. However, they do not rely on a gradient. Instead, they often use evolutionary algorithms (see, e.g., Eiben and Smith, 2008) to optimize the policy. Evolutionary algorithms find better solutions by combining (crossover) and slightly changing (mutation) already existing solutions. The solutions become better, because the probability of selecting an already existing solution to be modified and afterwards included in the successor generation depends on its quality (fitness). Mostly the fitness of a policy is determined by running it for a number of steps (policy roll-out). Usually, this is only feasible when a simulation is available, since many policies need to be evaluated for each generation and the optimization must run for a number of generations to come up with good solutions. Hence, the data-efficiency is often low. Nonetheless, policy search with evolutionary methods can lead to excellent policies. This is in particular true for non-Markovian environments, since they do not depend on the Markov property like value-function based approaches. A prominent example of methods used in this context is *neuroevolution*, where evolutionary algorithms are used to evolve the topology and weights of a neural control network (Gomez and Miikkulainen (1999); see Whiteson (2012) for a more recent treatment). *Fitted policy search* (Migliavacca et al., 2011) combines policy search and value functions; they evolve the parameters of the policy using a number of methods, including evolutionary algorithms, but determine the fitness not from actual roll-outs, but fitted policy evaluation with regression trees.

## 2.9 Exploration

Since RL often means learning from interaction with the environment, the aspect of *exploration* must be considered. Exploration in RL refers to the process of gathering information about the environment, i.e., about the consequences of different actions in different states.

A simple exploration scheme is *random exploration*, which is achieved by following the random policy that, regardless of the current state, chooses actions at

random. Naturally, the performance during exploration, i.e., the sum of collected rewards, is generally quite low compared to a policy that relies on already available knowledge about the environment.  $\varepsilon$ -greedy exploration tries to use already available knowledge while at the same time continuing to explore: with probability  $\varepsilon$ , a random action is chosen, with probability  $1 - \varepsilon$  the action which is so far considered best is chosen. Starting with a large  $\varepsilon$  and reducing it over time is therefore a simple approach to tackle the *exploration-exploitation dilemma*—when should one stop exploring and instead only execute actions assumed to be optimal? Similar to  $\varepsilon$ -greedy exploration is *Boltzmann action selection*, where the probability of selecting an action depends on its current  $Q$ -value estimate: the higher the estimated  $Q$ -value, the higher the probability of selecting that action. The influence of the  $Q$ -values depends on the Boltzmann temperature. The higher it is set, the smaller the  $Q$ -value’s influence and the more random the action selection. Similar to changing  $\varepsilon$  over time in  $\varepsilon$ -greedy exploration, it is reasonable to start with a high temperature and lower it over time.

One distinguishes directed and undirected exploration methods (Thrun, 1992). The approaches mentioned so far are undirected, since they do not plan the exploration, but explore randomly. Although  $\varepsilon$ -greedy and Boltzmann exploration explore “interesting” regions of the state space most of the time if  $\varepsilon$  or the Boltzmann temperature are sufficiently low, they are unable to systematically explore areas of the state space that are so far unexplored. This undirectedness leads to potentially exponential exploration times until a near-optimal policy is found. Directed methods, on the other hand, store not only information necessary to derive or represent the *exploitation* policy. In addition, they store information that allows more directed exploration.

A directed exploration method is *R-Max* (Brafman and Tenenbholz, 2003). It initializes the  $Q$ -function with the maximum possible value  $\frac{r_{\max}}{1-\gamma}$  for each state-action pair, where  $r_{\max}$  denotes the maximum possible immediate reward (hence the name). In the Bellman iteration the  $Q$ -value of a state-action pair is only updated if this state-action pair has been visited at least  $C$  times, with  $C$  a parameter of the algorithm. This way, each state-action pair appears to yield maximum return before it has been visited at least  $C$  times. When following the resulting policy, systematic exploration is achieved. Brafman and Tenenbholz (2003) showed that R-Max needs only polynomial time to find a near-optimal solution. This had first been shown by Kearns and Singh (1998) for their (more complicated)  $E^3$  algorithm.

Model-based interval estimation (MBIE) (Wiering and Schmidhuber, 1998; Strehl and Littman, 2009) is another algorithm for directed exploration. MBIE builds confidence intervals around the transition probability estimates as well as reward estimates. Then, in the Bellman iteration, the action is chosen which maximizes the  $Q$ -value within the intervals. Thus the action selection is maximally optimistic within the confidence intervals. When the policy is executed, new observations of the selected actions are collected, leading to smaller confidence intervals. If the

action selected so far is not clearly superior, the decreasing confidence interval will eventually lead to the selection of another action (whose confidence interval is still greater). Strehl and Littman (2009) showed that MBIE finds a near-optimal policy in polynomial time. They also introduced an MBIE variant called *model-based interval estimation with exploration bonus* (MBIE-EB), for which they also proved optimality (Strehl and Littman, 2009). MBIE-EB alters the update rule of the Bellman iteration to add a state-action dependent bonus that decreases with the number of observations of a state-action pair.

## 2.10 Further Reading

In this chapter a short introduction to RL was given. More details can be found in the excellent book by Sutton and Barto (1998). Another good introduction (that also includes POMDPs) is due to Kaelbling, Littman, and Moore (1996). The recent book by Szepesvári (2010) includes lots of methods developed in the past decade. Finally, Buşoniu, Babuška, Schutter, and Ernst (2010) treat RL and DP with a focus on function approximation.

# 3

## Uncertainty Awareness in Discrete Domains

This chapter presents methods for determining uncertainties of  $Q$ -values in discrete domains. The  $Q$ -values' uncertainties stem from the uncertainties about the parameters of the Markov decision process (MDP). This motivates the discussion of methods for deriving the  $Q$ -values' uncertainties from those of the MDP. Once the  $Q$ -values' uncertainties are known, they can be used for a number of applications that will be detailed in the next chapter. In the context of this work, the most important application is quality assurance, where the aim is to derive a *quantile-optimal* policy, whose performance distribution is narrower than that of the expectation-optimal one. Although the expected performance of a quantile-optimal policy may be lower than the expected performance of an expectation-optimal policy, the probability of obtaining a very low performance is reduced. Discrete methods are less affected by the problems mentioned in Section 1.1 of the introduction: for value iteration, there are no meta parameters to set and the learning process does not need to be monitored, since it deterministically arrives at the same results for a given MDP. However, if the MDP was estimated from observations, the need for the evaluation of the final policy still exists. It is only optimal w.r.t. the estimated MDP. One can only hope that it also performs sufficiently for the real MDP. Using uncertainty awareness for quality assurance means considering the uncertainty of the MDP estimate in a way that decreases the probability of obtaining a policy that will perform poor for the real MDP. This way good policies are generated more reliably.

### 3.1 Basic Idea

Classic value iteration based on dynamic programming (DP) determines the expectation of  $Q$ -values given an MDP. However, when the MDP, i.e., the transition probabilities  $\mathcal{P}$  and the reward function  $\mathcal{R}$ , are unknown, one can only *estimate* the MDP from observations. The resulting  $Q$ -values are optimal w.r.t. the estimated MDP, but not necessarily optimal w.r.t. the real MDP. Because of the

maximization of the  $Q$ -values done in the Bellman optimality equation, the resulting  $Q$ -function is positively biased and the resulting policy will often be too optimistic and perform insufficiently.

The basic idea of what will be done in this chapter is to consider not just point estimates of the MDP's parameters. Instead, for each parameter a distribution will be assumed, resulting in a distribution over MDPs. This distribution can then be used to determine a distribution of  $Q$ -values for each state-action pair instead of just a single value. Using this knowledge it becomes possible to determine *quantile-optimal* policies that try to maximize a certain quantile of the  $Q$ -value distribution instead of the expected value. A quantile-optimal policy optimizes the quantile performance instead of the expected (mean) performance. The lower the chosen quantile, the lower the probability of obtaining a policy that performs worse than indicated by the  $Q$ -values. The quantile realizes a trade-off between the amount of guaranteed performance and that guarantee's probability. The lower the quantile, the higher the probability of the guarantee, but the lower its value. It is a trade-off between performance and certainty. In quality assurance, one is willing to trade-in an amount of performance for a higher certainty that this performance is actually reached. It is crucial to note that we deal with the uncertainty of the estimators of the MDP, i.e., the uncertainty of having estimated the wrong MDP. This uncertainty is distinct from the stochasticity of the MDP.

In the following, a distribution is usually characterized only by its expected value and standard deviation.

First possibilities of how to model the transition probabilities and the rewards of an MDP are mentioned. It is then illustrated how to approximate the  $Q$ -value distribution by repeatedly drawing an MDP  $M_i$  from the MDP distribution and determining the  $Q$ -function for  $M_i$  (Monte Carlo uncertainty estimate (Section 3.3)). Next, work by Schneegaß, Udluft, and Martinetz (2008) is described. Schneegaß, Udluft, and Martinetz (2008) used uncertainty propagation (UP) to calculate the  $Q$ -function's uncertainty directly from the uncertainties of the estimators. This makes it possible to determine the exact uncertainties without the need for computationally expensive Monte Carlo methods. However, the computational burden of this method is still high. Therefore, an approximation is presented whose computational complexity is identical to that of the standard Bellman iteration. Nonetheless, the uncertainties determined by this approximation can be used for quality assurance and exploration, which will be shown experimentally in the next chapter.

## 3.2 Estimators

There are several ways of modeling the estimators for the transition probabilities  $\mathcal{P}$  and the reward  $\mathcal{R}$ . In the following, the frequentist approach using relative

frequency as well as a Bayesian approach are presented.

For both approaches the probabilities from state-action pairs are assumed to be independent of each other and the rewards. One can therefore assume an independent multinomial distribution for each state-action pair. For a state-action pair  $(s, a)$ , the probability of a successor state is then given as  $\hat{P}(s'|s, a)$ , with  $\sum_{s' \in \mathcal{S}} \hat{P}(s'|s, a) = 1$ .

### 3.2.1 Frequentist Estimate

In the frequentist paradigms, the relative frequency is used as the expected transition probability, i.e.,

$$\hat{\mathcal{P}}(s_k | s_i, a_j) = \frac{n_{s_i, a_j}}{n_{s_k | s_i, a_j}}, \quad (3.1)$$

where  $n_{s_i, a_j}$  denotes the total number of transitions from state-action pair  $(s_i, a_j)$  of which  $n_{s_k | s_i, a_j}$  lead to state  $s_k$ .

The uncertainties of the parameters of the according multinomial distribution, represented by a covariance matrix, are assumed to be

$$\mathbf{Cov}(\hat{\mathcal{P}}(s_k | s_i, a_j), \hat{\mathcal{P}}(s_n | s_i, a_j)) = \begin{cases} \frac{\hat{\mathcal{P}}(s_k | s_i, a_j)(1 - \hat{\mathcal{P}}(s_k | s_i, a_j))}{n_{s_i, a_j} - 1}, & \text{if } s_k = s_n, \\ \frac{-\hat{\mathcal{P}}(s_k | s_i, a_j)\hat{\mathcal{P}}(s_n | s_i, a_j)}{n_{s_i, a_j} - 1}, & \text{if } s_k \neq s_n. \end{cases} \quad (3.2)$$

Note that since each state-action pair has its own multinomial distribution, it has also its own corresponding covariance matrix.

Using the same concept for the rewards and assuming a normal distribution, the mean is used as reward expectation, i.e.,

$$\hat{\mathcal{R}}(s_i, a_j, s_k) = \frac{1}{n_{s_k | s_i, a_j}} \sum_{i=1}^{n_{s_k | s_i, a_j}} r_{n_{s_k | s_i, a_j}}^{(i)}, \quad (3.3)$$

with  $r_{n_{s_k | s_i, a_j}}^{(i)}$  the  $i$ -th observation of the reward of the transition  $s_i \rightarrow_{a_j} s_k$ .

The corresponding covariance matrix is diagonal with elements

$$\mathbf{Cov}(\hat{\mathcal{R}}(s_i, a_j, s_k), \hat{\mathcal{R}}(s_i, a_j, s_k)) = \frac{\mathbf{Var}(\hat{\mathcal{R}}(s_i, a_j, s_k))}{n_{s_k | s_i, a_j} - 1}. \quad (3.4)$$

Although the estimation of the transition probabilities using relative frequency usually leads to good results in practice, the corresponding uncertainty estimation is problematic if there are only a few observations, because in that case the uncertainties are often underestimated. For instance, if a specific transition is observed twice out of two tries ( $n_{s_k | s_i, a_j} = n_{s_i, a_j} = 2$ ), its uncertainties are assumed to be  $\mathbf{Cov}(\hat{\mathcal{P}}(s_k | s_i, a_j), \hat{\mathcal{P}}(s_k | s_i, a_j)) = 0$ .

### 3.2.2 Bayesian Estimate

Again assuming all transitions from different state-action pairs to be independent of each other and the rewards, the transitions can be modeled as multinomial distributions. In a Bayesian setting, where one assumes a prior distribution over the parameter space  $\mathcal{P}(s_k|s_i, a_j)$  for given  $i$  and  $j$ , the Dirichlet distribution with density

$$\Pr(\mathcal{P}(s_1|s_i, a_j), \dots, \mathcal{P}(s_{|\mathcal{S}}|s_i, a_j))_{\alpha_{ij1}, \dots, \alpha_{ij|\mathcal{S}}}} = \frac{\Gamma(\alpha_{ij})}{\prod_{k=1}^{|\mathcal{S}|} \Gamma(\alpha_{ijk})} \prod_{k=1}^{|\mathcal{S}|} \mathcal{P}(s_k|s_i, a_j)^{\alpha_{ijk}-1}, \quad (3.5)$$

$\alpha_{ij} = \sum_{k=1}^{|\mathcal{S}|} \alpha_{ijk}$ , is a conjugate prior with posterior parameters  $\alpha_{ijk}^d = \alpha_{ijk} + n_{s_i a_j s_k}$ ,  $\alpha_{ij}^d = \sum_{k=1}^{|\mathcal{S}|} \alpha_{ijk}^d$ . Choosing the expectation of the posterior distribution as the estimator, i.e.,

$$\hat{\mathcal{P}}(s_k|s_i, a_j) = \frac{\alpha_{ijk}^d}{\alpha_{ij}^d}, \quad (3.6)$$

the uncertainty of  $\hat{\mathcal{P}}$  is

$$\mathbf{Cov}(\hat{\mathcal{P}}(s_k|s_i, a_j), \hat{\mathcal{P}}(s_l|s_i, a_j)) = \begin{cases} \frac{\alpha_{i,j,k}^d(1-\alpha_{i,j,k}^d)}{(\alpha_{i,j}^d)^2(\alpha_{i,j}^d+1)}, & \text{if } s_k = s_l, \\ \frac{-\alpha_{i,j,k}^d \alpha_{i,j,l}^d}{(\alpha_{i,j}^d)^2(\alpha_{i,j}^d+1)} & \text{if } s_k \neq s_l. \end{cases} \quad (3.7)$$

Note that  $\alpha_i = 0$  results in a prior that leads to the same estimates and slightly lower uncertainties compared to the frequentist modeling of Section 3.2.1. On the other hand, setting  $\alpha_i = 1$  leads to a flat, maximum entropy prior that assumes all transitions from a state to all other states equally probable.

Both settings,  $\alpha_i = 0$  and  $\alpha_i = 1$ , represent extremes that the author believes are unreasonable for most applications. Instead, the prior belief is modeled by setting  $\alpha_i = \frac{m}{|\mathcal{S}|}$ , where  $m$  is the average number of expected successor states of all state-action pairs and  $|\mathcal{S}|$  is the total number of states. This choice of  $\alpha_i$  realizes an approximation of a maximum entropy prior over a subset of the state space with a size of  $m$  states. This way most of the probability is “distributed” among any subset of  $m$  states that have actually been observed, the probability of all other (not observed) successor states becomes very low. Compared to the maximum entropy prior with  $\alpha_i = 1$ , one needs only a few observations for the actually observed successor states to be much more probable than not observed ones. At the same time, the estimation of the uncertainty is not as extreme as the frequentist one, since having made the same observation twice does not cause the uncertainty to become zero. Estimating  $m$  from the observations is easily possible.

Friedman and Singer (1999) propose to use a hierarchical prior to achieve some-

thing similar. With their approach they try to distribute the probability mass among a relatively low number of actually observed outcomes compared to the number of possible outcomes. Instead of directly estimating  $\alpha = \frac{m}{|\mathcal{S}|}$ , one first uses a prior over the feasible sets of possible outcomes.

### 3.3 Monte Carlo Uncertainty Estimate

As noted before, we are looking for a distribution of  $Q$ -values instead of just single point estimates. The previous sections described ways to estimate MDPs from observations. In fact, not just an MDP is estimated, but a distribution of MDPs. Using this distribution, a straightforward solution to the problem of estimating a  $Q$ -value distribution is a Monte Carlo approach (Metropolis and Ulam, 1949): One samples a number of MDPs from the MDP distribution and calculates the  $Q$ -values for each of them using value iteration. For a state-action pair, each MDP sample gives one value. Those values combined approximate the true  $Q$ -value distribution for that state-action pair. Obviously, the more MDPs one samples and calculates  $Q$ -values for, the better the approximation will be. Something quite similar to this approach was done by Dearden, Friedman, and Andre (1999) for efficient exploration. They used the uncertainty to direct the exploration to promising but still insufficiently state-action pairs. In the present work, however, the notion of uncertainty is used in a more explicit sense and the application is not limited to exploration. Moreover, the Monte Carlo sampling approach is just one way to estimate the  $Q$ -value distribution. It is discussed first as it is the most straightforward one. Its understanding is therefore a good basis for the further discussion.

It is assumed that the Bayesian estimator is used. Thus, in order to sample an MDP, one needs to sample from the Dirichlet posterior distributions. This can be achieved with independent samples from a Gamma distribution and a normalization (Congdon, 2006, p. 83). Since the state-action pairs are assumed to be independent of each other, it is possible to sample a multinomial distribution for each state-action pair independently. Thus, for a state-action pair  $(s_i, a_j)$ , the according multinomial distribution is sampled the following way:

$$p_1 \sim \Gamma(\alpha_{i,j,1}^d), p_2 \sim \Gamma(\alpha_{i,j,2}^d), \dots, p_{|\mathcal{S}|} \sim \Gamma(\alpha_{i,j,|\mathcal{S}|}^d), \quad (3.8)$$

$$\mathcal{P}(s_1|s_i, a_j) := \frac{p_1}{\sum_{i=1}^{|\mathcal{S}|} p_i}, \mathcal{P}(s_2|s_i, a_j) := \frac{p_2}{\sum_{i=1}^{|\mathcal{S}|} p_i}, \dots, \mathcal{P}(s_{|\mathcal{S}|}|s_i, a_j) := \frac{p_{|\mathcal{S}|}}{\sum_{i=1}^{|\mathcal{S}|} p_i}. \quad (3.9)$$

Doing this for each state-action pair it is possible to sample an MDP  $M_k$  from the posterior distribution.

This way one samples a number of  $K$  MDPs  $M_k, k = 1, 2, \dots, K$ , determines

**Algorithm 3:** Monte Carlo Uncertainty Estimation**Input:**  $\alpha_{i,j,k}, i = 1, 2, \dots, |\mathcal{S}|, j = 1, 2, \dots, |\mathcal{A}|, k = 1, 2, \dots, |\mathcal{S}|, K$ **Result:**  $Q_k^*, k = 1, 2, \dots, K$ 

```

1 begin
  | sample MDPs
2   for  $k = 1, 2, \dots, K$  do
3     for  $\forall (s, a) \in S \times \mathcal{A}$  do
4       | for  $s' \in S$  do
5         | |  $p_{s'} \sim \Gamma(\alpha_{i,j,k}^d)$ 
6       | for  $s' \in S$  do
7         | |  $\mathcal{P}_i(s'|s, a) \leftarrow \frac{p_{s'}}{\sum_{s \in S} p_s}$ 
8     determine expectation-optimal policy
9      $\pi^* \leftarrow \text{value\_iteration}(\mathbf{E}_i \{ \mathcal{P}_i \}, \mathcal{R})$ 
10    determine  $\pi^*$ 's  $Q$ -functions for sampled MDPs
11    for  $i = 1, 2, \dots, k$  do
12      |  $Q_i^* \leftarrow \text{policy\_evaluation}(\pi^*, \mathcal{P}_i, \mathcal{R})$ 
13    return  $Q_i^*, i = 1, 2, \dots, k$ 

```

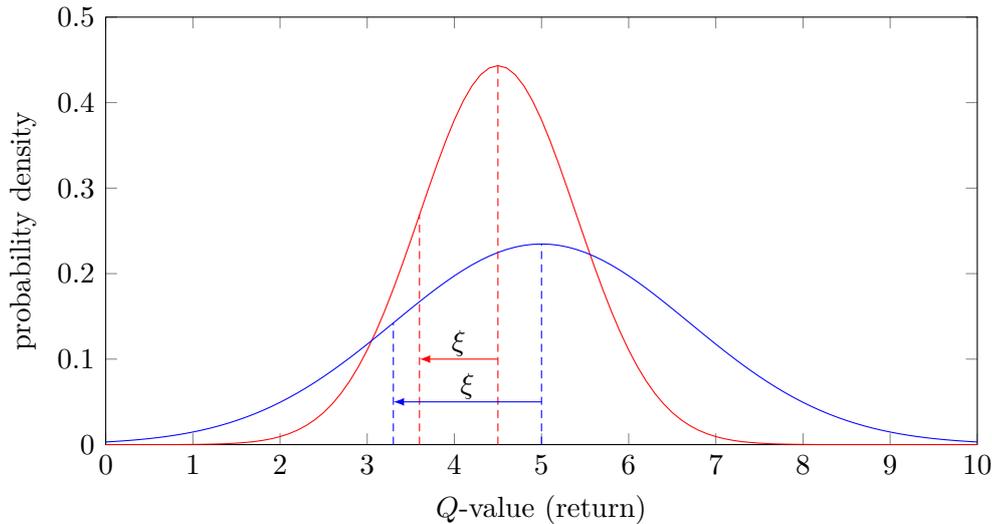
the  $Q$ -function for each MDP  $M_i$  with value iteration, and arrives at  $K$  values  $Q_i^*(s, a)$  for each state-action pair  $(s, a)$ , approximating the distribution of this particular state-action pair's  $Q$ -value.

Unfortunately, the method derived so far is inconsistent. Each  $Q$ -function implies a certain policy and is valid only w.r.t. that policy. Combining them to approximate the  $Q$ -value distribution would mean mixing  $Q$ -functions from different policies. Moreover, what would be the policy that this  $Q$ -value distribution belongs to? A more consistent solution could look like the following:

1. Determine a common policy  $\pi$  for the MDP distribution for the expected MDP, i.e., using the expected values of the corresponding Dirichlet posterior distributions.
2. Sample  $K$  MDPs  $M_k, k = 1, 2, \dots, K$  from the MDP distribution.
3. For each MDP, do policy evaluation (Section 2.4.1) for the common policy  $\pi$  to get  $Q_k^\pi$ , the  $Q$ -function of  $\pi$  evaluated for MDP  $M_k$ .

Algorithm 3 summarizes Monte Carlo uncertainty estimation of the expectation-optimal  $Q$ -function.

With the method one obtains an array of  $Q$ -values  $q_i^{s_i, a_j}, i = 1, 2, \dots, K$  for each state-action pair  $(s_i, a_j)$ , from which the distribution can be approximated. For



**Figure 3.1:** Exemplary return distributions of two actions (red and blue) in a state. The blue action gives the higher expected return, therefore a expectation-optimal policy would choose this action. However, if one considers the uncertainty as well, here with  $\xi = 1$ , the red action becomes preferable, because its return is more certain.

instance, the mean value gives the expected  $Q$ -value,

$$\mathbf{E} \{Q^\pi(s_i, a_j)\} := \frac{1}{K} \sum_{i=1}^K q_i^{s_i, a_j}, \quad (3.10)$$

the standard deviation of the distribution is approximated by the standard deviation of the array,

$$\sigma Q^\pi(s_i, a_j) := \sqrt{\mathbf{Var}(q^{s_i, a_j})}. \quad (3.11)$$

With  $K \rightarrow \infty$ ,  $\mathbf{E} \{Q^\pi(s_i, a_j)\} \rightarrow Q^\pi(s_i, a_j)$ .

This can be used to reason about the true return of  $\pi$ . Given the uncertainty  $\sigma Q^\pi$ , we can define a function

$$Q_u^{\pi, \xi}(s, a) = Q^\pi(s, a) - \xi \sigma Q(s, a) \quad (3.12)$$

that gives the return guaranteed (performance limit) with some probability depending on  $\xi$  and the distribution class of  $Q^\pi$ . For instance, assuming that  $Q^\pi$  is normally distributed,  $Q_u^{\pi, 2}$  ( $\xi = 2$ ) gives the minimum return that with probability  $\Pr(\xi) = \Pr(2) \approx 0.977$  will occur.

Figure 3.1 illustrates the situation of two actions selectable in a given state. The blue action has the higher expected return, but in comparison to the red action its return distribution is quite wide, hence the probability of obtaining a low return in one of the possible MDPs is higher. When optimizing the expected value, one would choose the blue action and with probability  $\Pr(0) = 0.5$  obtain a return of

at least 5. The red action obtains with probability 0.5 a return of at least 4.5. In the example a choice of  $\xi = 1$  is illustrated as well. With a probability of  $\Pr(\xi = 1) \approx 0.84$  the blue action will generate a return of 3.3. The red action will with the same probability obtain a return of 3.6. A quantile-optimal policy determined with  $\xi = 1$  would thus choose the red action.

### 3.4 Uncertainty-Aware Value Iteration

So far we have only estimated the uncertainty of a given policy or  $Q$ -function. The next step is to maximize a certain performance quantile defined by  $\xi$ . This is necessary because simply changing the policy using the uncertainty of its  $Q$ -function does not work: A policy based on  $Q_u^{\pi, \xi}$ , i.e.,  $\pi_u^\xi(s) := \arg \max_{a \in \mathcal{A}} Q_u^{\pi, \xi}$ , does not in general improve the performance limit, as  $Q_u^{\pi, \xi}$  considers the uncertainty only for one step.  $Q_u^{\pi, \xi}$  gives the quantile performance of  $\pi$ . If one changes the policy, the existing  $Q$ -function would become invalid, posing an inconsistency. To use the knowledge of uncertainty for maximizing the performance limit (as opposed to the expectation), the uncertainty needs to be incorporated into the policy-improvement step of the value iteration algorithm.

The policy-improvement step is contained within the Bellman optimality equation as  $\max_{a \in \mathcal{A}} Q^m(s, a)$ . Alternatively, determining the optimal policy in each iteration as

$$\forall s : \pi^m(s) \leftarrow \arg \max_{a \in \mathcal{A}} Q^m(s, a) \quad (3.13)$$

and then updating the  $Q$ -function using this policy, i.e.,

$$\forall s, a : Q^m(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \hat{\mathcal{P}}(s'|s, a) \left[ \hat{\mathcal{R}}(s, a, s') + \gamma Q^{m-1}(s', \pi^{m-1}(s)) \right], \quad (3.14)$$

yields the same solution. To determine a *quantile-* or  $\xi$ -*optimal* policy that maximizes the performance limit for a given  $\xi$ , the update of the policy must not choose the optimal action w.r.t. the maximum over the  $Q$ -values of a particular state, but the maximum over the  $Q$ -values minus their weighted uncertainty:

$$\forall s : \pi^m(s) \leftarrow \arg \max_{a \in \mathcal{A}} [Q^m(s, a) - \xi \sigma Q^m(s, a)]. \quad (3.15)$$

The Monte Carlo uncertainty estimation approach can be modified to determine a quantile-optimal policy by considering the uncertainty in the policy improvement step of value iteration. In each iteration, the policy is determined according to (3.15), where  $Q^m$  and  $\sigma Q^m$  are estimated from the samples. See Algorithm 4 for the complete procedure. In the policy improvement step in line 13,  $\frac{1}{K} \sum_i Q_i^m(s, a)$  corresponds to  $Q^m(s, a)$ ,  $\sqrt{\mathbf{Var}_i(Q_i^m(s, a))}$  corresponds to  $\sigma Q^m(s, a)$ .

We now have everything we need to derive policies that optimize a specific quantile, i.e., policies maximizing  $Q_u^{\pi, \xi}$  for a given  $\xi$ . With  $\xi > 0$  we get policies

**Algorithm 4:** Monte Carlo Uncertainty-Aware Value Iteration**Input:**  $\alpha_{i,j,k}, i = 1, 2, \dots, |\mathcal{S}|, j = 1, 2, \dots, |\mathcal{A}|, K = 1, 2, \dots, |\mathcal{S}|, K, \xi$ **Result:**  $Q_i^*, i = 1, 2, \dots, K$ 

```

1 begin
  sample MDPs
2 for  $i = 1, 2, \dots, k$  do
3   for  $\forall (s, a) \in \mathcal{S} \times \mathcal{A}$  do
4     for  $s' \in \mathcal{S}$  do
5        $p_{s'} \sim \Gamma(\alpha_{i,j,k}^d)$ 
6     for  $s' \in \mathcal{S}$  do
7        $\mathcal{P}_i(s'|s, a) \leftarrow \frac{p_{s'}}{\sum_{s \in \mathcal{S}} p_s}$ 
  uncertainty-aware value iteration
8  $m \leftarrow 0$ 
9 for  $i = 1, 2, \dots, K$  do
10    $Q_i^0 \leftarrow 0$ 
11 while not converged do
12   determine this iteration's policy
13   for  $s, a \in \mathcal{S} \times \mathcal{A}$  do
14      $\pi^m(s) \leftarrow \arg \max_{a \in \mathcal{A}} \frac{1}{K} \sum_i Q_i^m(s, a) - \xi \sqrt{\mathbf{Var}_i(Q_i^m(s, a))}$ 
15   do value iteration update for each MDP sample
16   for  $i = 1, 2, \dots, K$  do
17     for  $s, a \in \mathcal{S} \times \mathcal{A}$  do
18        $Q_i^{m+1}(s, a) \leftarrow \sum_{s'} \mathcal{P}_i(s'|s, a) [\mathcal{R}(s, a, s') + \gamma Q_i^m(s', \pi^m(s'))]$ 
19    $m \leftarrow m + 1$ 
20 return  $Q_i^*, i = 1, 2, \dots, K$ 

```

avoiding uncertainty that can be used for quality assurance, while with  $\xi < 0$  we get policies that seek uncertainty and can be used for efficient exploration. Before detailing those applications with experiments in the next chapter, more direct ways of estimating the uncertainty will be discussed in the following sections.

### 3.5 Full-Matrix Uncertainty Propagation

The estimators described in Section 3.2 give expected values and uncertainties. Instead of sampling MDPs, one can use the uncertainties directly to compute the uncertainty of the  $Q$ -function. This can be achieved by applying uncertainty propagation (UP) to the Bellman iteration to derive an update equation that will be used in parallel to the standard Bellman update equation.

Uncertainty propagation, also known as Gaussian error propagation (D'Agostini, 2003), is a common method in statistics to propagate the uncertainty of measurements to the results. It is based on a first-order Taylor expansion. Given a function  $f(x)$  with  $f : \mathbb{R}^M \mapsto \mathbb{R}^N$  and the uncertainty of the function arguments as covariance matrix  $\mathbf{Cov}(x)$ , the uncertainty of the function values  $f(x)$  is determined as

$$\mathbf{Cov}(f) = \mathbf{Cov}(f, f) = D\mathbf{Cov}(x)D^T. \quad (3.16)$$

$D$  is the Jacobian matrix of  $f$  w.r.t.  $x$  consisting of the partial derivatives of  $f$  w.r.t. each component of  $x$ , i.e.,  $D_{i,j} = \frac{\partial f_i}{\partial x_j}$ .

We now want to use the knowledge about the estimator's uncertainty to determine the  $Q$ -function's uncertainty,  $\sigma Q$ . We start with an initial covariance matrix  $\mathbf{Cov}(Q^0, \mathcal{P}, \mathcal{R})$  and apply UP to the update equation of the Bellman iteration

$$Q^m(s, a) \leftarrow \sum_{s'} \hat{\mathcal{P}}(s'|s, a) \left[ \hat{\mathcal{R}}(s, a, s') + \gamma V^{m-1}(s') \right], \quad (3.17)$$

where  $\hat{\mathcal{P}}$  and  $\hat{\mathcal{R}}$  denote the estimates for the transition probabilities and rewards, respectively, and  $V^{m-1}$  the value function of the previous iteration. The Bellman equation takes  $\hat{\mathcal{P}}$ ,  $\hat{\mathcal{R}}$ , and  $V^{m-1}$  as arguments and produces  $Q^m$  as result. To apply uncertainty propagation, equation (3.16) is used on the Bellman equation to obtain the update rule

$$\mathbf{Cov}(Q^m, \mathcal{P}, \mathcal{R}) \leftarrow D^{m-1} \mathbf{Cov}(Q^{m-1}, \mathcal{P}, \mathcal{R}) (D^{m-1})^T, \quad (3.18)$$

where  $D^m$  denotes the Jacobian matrix (obtained by differentiating equation (3.17))

$$D^m = \begin{pmatrix} D_{Q,Q}^m & D_{Q,\mathcal{P}}^m & D_{Q,\mathcal{R}}^m \\ 0 & \mathbf{I} & 0 \\ 0 & 0 & \mathbf{I} \end{pmatrix} \quad (3.19)$$

$$\begin{aligned} (D_{Q,Q}^m)_{(i,j),(k,l)} &= \gamma \pi^m(s_k, a_l) \hat{\mathcal{P}}(s_k | s_i, a_j) \\ (D_{Q,\mathcal{P}}^m)_{(i,j),(l,n,k)} &= \delta_{i,l} \delta_{j,n} \left( \hat{\mathcal{R}}(s_i, a_j, s_k) + \gamma V^m(s_k) \right) \\ (D_{Q,\mathcal{R}}^m)_{(i,j),(l,n,k)} &= \delta_{i,l} \delta_{j,n} \hat{\mathcal{P}}(s_k | s_i, a_j). \end{aligned}$$

Note that in the above definition of  $D^m$  a stochastic policy  $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$  is assumed that gives the probability of choosing action  $a$  in state  $s$ .<sup>1</sup> Starting with

<sup>1</sup>A deterministic policy  $\pi_d$  can easily be mapped to a stochastic one by setting  $\pi(s, a) := 1$  if  $\pi_d(s) = a$  and  $\pi(s, a) := 0$  otherwise.

an initial covariance matrix

$$\mathbf{Cov}(Q^0, \mathcal{P}, \mathcal{R}) = \begin{pmatrix} \mathbf{Cov}(Q^0) & \mathbf{Cov}(Q^0, \mathcal{P}) & \mathbf{Cov}(Q^0, \mathcal{R}) \\ \mathbf{Cov}(\mathcal{P}, Q^0) & \mathbf{Cov}(\mathcal{P}) & \mathbf{Cov}(\mathcal{P}, \mathcal{R}) \\ \mathbf{Cov}(\mathcal{R}, Q^0) & \mathbf{Cov}(\mathcal{P}, \mathcal{R})^T & \mathbf{Cov}(\mathcal{R}) \end{pmatrix} \quad (3.20)$$

$$= \begin{pmatrix} 0 & 0 & 0 \\ 0 & \mathbf{Cov}(\mathcal{P}) & \mathbf{Cov}(\mathcal{P}, \mathcal{R}) \\ 0 & \mathbf{Cov}(\mathcal{P}, \mathcal{R})^T & \mathbf{Cov}(\mathcal{R}) \end{pmatrix}, \quad (3.21)$$

the update rule (3.18) is used in parallel to the Bellman update equation in each iteration to update the covariance matrix.

Finally, from the covariance matrix one can extract the  $Q$ -function's uncertainty as  $\sigma Q^m := \sqrt{\text{diag}(\mathbf{Cov}(Q^m))}$ .

## 3.6 Efficient Diagonal Approximation

The above algorithm's time complexity per iteration is of higher order than the standard Bellman iteration's one, which needs  $O(|\mathcal{S}|^2|\mathcal{A}|)$  time ( $O(|\mathcal{S}|^2|\mathcal{A}|^2)$  for stochastic policies). The bottleneck is the covariance update with a time complexity of  $O((|\mathcal{S}||\mathcal{A}|)^{2.376})$  (Coppersmith and Winograd, 1990), since each entry of  $Q$  depends only on  $|\mathcal{S}|$  entries of  $\mathcal{P}$  and  $\mathcal{R}$ . The overall complexity is hence bounded by these magnitudes.

This complexity can limit the applicability of the algorithm for problems with more than a few hundred states. To circumvent this issue, it is possible to use an approximate version of the algorithm that considers only the diagonal of the covariance matrix. This variant is called the diagonal approximation of uncertainty-incorporating policy iteration (DUIPI). Only considering the diagonal neglects the correlations between the state-action pairs, which in fact are small for many RL problems, where on average different state-action pairs share only little probability to reach the same successor state.

DUIPI is easier to implement and, most importantly, has the same complexity as the standard Bellman iteration—both in terms of time and space complexity. In the following we will derive the update equations for DUIPI.

When neglecting correlations, the uncertainty of values  $f(x)$  with  $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ , given the uncertainty of the arguments  $x$  as  $\sigma x$ , is determined as

$$(\sigma f)^2 = \sum_i \left( \frac{\partial f}{\partial x_i} \right)^2 (\sigma x_i)^2. \quad (3.22)$$

This is equivalent to equation (3.16) of full-matrix UP with all non-diagonal elements set equal to zero.

Recall the update step of the Bellman iteration,

$$Q^m(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) \left[ \mathcal{R}(s, a, s') + \gamma V^{m-1}(s') \right], \quad (3.23)$$

can be regarded as a function of the estimated transition probabilities  $\mathcal{P}$  and rewards  $\mathcal{R}$ , and the  $Q$ -function of the previous iteration  $Q^{m-1}$  ( $V^{m-1}$  is a subset of  $Q^{m-1}$ ), that yields the updated  $Q$ -function  $Q^m$ . Applying UP as given by equation (3.22) to the Bellman iteration, one obtains an update equation for the  $Q$ -function's uncertainty:

$$\begin{aligned} (\sigma Q^m(s, a))^2 \leftarrow & \sum_{s' \in \mathcal{S}} (D_{Q,Q})^2 (\sigma V^{m-1}(s'))^2 + \\ & \sum_{s' \in \mathcal{S}} (D_{Q,\mathcal{P}})^2 (\sigma \mathcal{P}(s'|s, a))^2 + \\ & \sum_{s' \in \mathcal{S}} (D_{Q,\mathcal{R}})^2 (\sigma \mathcal{R}(s, a, s'))^2, \end{aligned} \quad (3.24)$$

$$D_{Q,Q} = \gamma \mathcal{P}(s'|s, a), \quad D_{Q,\mathcal{P}} = \mathcal{R}(s, a, s') + \gamma V^{m-1}(s'), \quad D_{Q,\mathcal{R}} = \mathcal{P}(s'|s, a). \quad (3.25)$$

$V^m$  and  $\sigma V^m$  have to be set depending on the desired type of the policy (stochastic or deterministic) and whether policy evaluation or policy iteration is performed. E.g., for policy evaluation of a stochastic policy  $\pi$

$$V^m(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^m(s, a), \quad (3.26)$$

$$(\sigma V^m(s))^2 = \sum_{a \in \mathcal{A}} \pi(a|s)^2 (\sigma Q^m(s, a))^2. \quad (3.27)$$

For policy iteration, according to the Bellman optimality equation and resulting in the  $Q$ -function  $Q^*$  of an optimal policy,  $V^m(s) = \max_a Q^m(s, a)$  and  $(\sigma V^m(s))^2 = (\sigma Q^m(s, \arg \max_a Q^m(s, a)))^2$ .

Using the estimates  $\hat{\mathcal{P}}$  and  $\hat{\mathcal{R}}$  with their uncertainties  $\sigma \hat{\mathcal{P}}$  and  $\sigma \hat{\mathcal{R}}$  and starting with an initial  $Q$ -function  $Q^0$  and corresponding uncertainty  $\sigma Q^0$ , e.g.,  $Q^0 \leftarrow 0$  and  $\sigma Q^0 \leftarrow 0$ , through the update equations (3.23) and (3.24) the  $Q$ -function and corresponding uncertainty are updated in each iteration and converge to  $Q^\pi$  and  $\sigma Q^\pi$  for policy evaluation and  $Q^*$  and  $\sigma Q^*$  for policy iteration. Algorithm 5 shows the complete DUIPI procedure.

Instead of calculating the  $Q$ -function and its uncertainty and then using this information to update the policy, one can as well modify the  $Q$ -values by adding or subtracting the  $\xi$ -weighted uncertainty in each iteration. This gives rise to a variant of DUIPI called the diagonal approximation of uncertainty-incorporating policy iteration with  $Q$  modification (DUIPI-QM). However, this leads to a  $Q$ -function that is no longer the  $Q$ -function of the policy, as it contains not only

**Algorithm 5:** DUIPI

**Input:** estimators  $\mathcal{P}$  and  $\mathcal{R}$  for a discrete MDP, their uncertainties  $\sigma\mathcal{P}$  and  $\sigma\mathcal{R}$ ,  
a scalar  $\xi$

**Result:** quantile-optimal policy  $\pi$

```

1 begin
   Initialize  $Q$ -function and uncertainty with zero
2    $Q^0 \leftarrow 0$ 
3    $(\sigma Q)^2 \leftarrow 0$ 
   initialize policy to choose actions with equal probabilities
4    $\pi \leftarrow \frac{1}{|\mathcal{A}|}$ 
5    $m \leftarrow 0$ 
6   while the desired precision is not reached do
   Update policy and value function
7   for  $s \in \mathcal{S}$  do
   determine best action
8    $a_{s,\max} \leftarrow \arg \max_a Q^m(s, a) - \xi \sqrt{(\sigma Q^m)^2(s, a)}$ 
   increase probability of choosing that action ...
9    $d_s \leftarrow \min(1/m, 1 - \pi^m(a_{s,\max}|s))$ 
10   $\pi^{m+1}(a_{s,\max}|s) \leftarrow \pi^m(a_{s,\max}|s) + d_s$ 
   ... and decrease probability for all other actions
11  for  $a \in \mathcal{A} \setminus \{a_{s,\max}\}$  do
12   $\pi^{m+1}(a|s) \leftarrow \frac{1 - \pi^m(a_{s,\max}|s)}{1 - \pi^m(a_{s,\max}|s) + d_s} \pi^m(a|s)$ 
   Update value function and its uncertainty
13   $V^{m+1}(s) \leftarrow \sum_a \pi^{m+1}(s, a) Q^m(s, a)$ 
14   $(\sigma V^{m+1})^2(s) \leftarrow \sum_a \pi^{m+1}(s, a) (\sigma Q^m)^2(s, a)$ 
   update  $Q$ -function and its uncertainty
15  for  $s, a \in \mathcal{S} \times \mathcal{A}$  do
16   $Q^{m+1}(s, a) \leftarrow \sum_{s'} \mathcal{P}(s'|s, a) [\mathcal{R}(s, a, s') + \gamma V^{m+1}(s')]$ 
17   $(\sigma Q^{m+1})^2(s, a) \leftarrow \sum_{s'} (D_{Q,Q})^2 (\sigma V^{m+1})^2(s') +$ 
18   $(D_{Q,\mathcal{P}})^2 (\sigma \mathcal{P})^2(s'|s, a) + (D_{Q,\mathcal{R}})^2 (\sigma \mathcal{R})^2(s, a, s')$ 
19   $m \leftarrow m + 1$ 
20 return  $\pi$ 

```

the sum of (discounted) rewards, but also uncertainties. Therefore, using this  $Q$  and  $\sigma Q$  it is not possible to reason about expected rewards and uncertainties when following this policy. Moreover, for the exploration case with negative  $\xi$  the  $Q$ -function does not converge in general for this update scheme, because in each iteration the  $Q$ -function is increased by the  $\xi$ -weighted uncertainty, which in turn leads to higher uncertainties in the next iteration. On the other hand, by choosing  $\xi$  and  $\gamma$  to satisfy  $\xi + \gamma < 1$  it is possible to keep  $Q$  and  $\sigma Q$  from

diverging. Nonetheless, in the experiments reported in the next chapter DUIPI-QM has proven useful, as its update scheme allows to use DUIPI successfully even for environment that exhibit high correlations between different state-action pairs, because by updating the  $Q$ -values the uncertainty is propagated through them.

Like the full-matrix algorithm DUIPI and DUIPI-QM can be used with any choice of estimator, e.g., a Bayesian setting using Dirichlet priors or the frequentist paradigm (see Section 3.2). The only requirement is the possibility to access the estimate's uncertainties  $\sigma\mathcal{P}$  and  $\sigma\mathcal{R}$ .

Algorithm	Time Complexity	Space Complexity
<b>Full-Matrix UP</b>	$O(( \mathcal{S}  \mathcal{A} )^{2.376})$	$O( \mathcal{S} ^5 \mathcal{A} ^3)$
<b>DUIPI</b>	$O( \mathcal{S} ^2 \mathcal{A} )$	$O( \mathcal{S} ^2 \mathcal{A} )$
<b>DUIPI-QM</b>	$O( \mathcal{S} ^2 \mathcal{A} )$	$O( \mathcal{S} ^2 \mathcal{A} )$
<b>Standard Value Iteration</b>	$O( \mathcal{S} ^2 \mathcal{A} )$	$O( \mathcal{S} ^2 \mathcal{A} )$

**Table 3.1:** Time and space complexities of the algorithms.

### 3.7 Summary

This chapter introduced methods for determining a  $Q$ -function's uncertainty in problems with discrete state and action spaces. Once the uncertainty is known, it can be used to derive quantile-optimal policies that optimize a quantile performance instead of the expectation. First, the frequentist and a Bayesian approach to model an MDP distribution were described. Next, a Monte Carlo approach to estimate the  $Q$ -function's uncertainty and derive quantile-optimal policies (defined by a parameter  $\xi$ ) was proposed. The Monte Carlo approach illustrates the relationship of model uncertainty (w.r.t. the MDP) and resulting uncertainty of the  $Q$ -function. It further has the advantage of actually approximating the true  $Q$ -value distribution. Its downside is the computational cost. The rest of the chapter dealt with ways of directly estimating the uncertainty from the expected value and standard deviation of the MDP distributions. The full-matrix uncertainty propagation approach (Schneegaß, Udluft, and Martinetz, 2008) was described. To deal with this method's shortcomings in terms of computational burden, an efficient approximation (DUIPI) was proposed. Further, a variation that modifies the  $Q$ -values using their uncertainty (DUIPI-QM) was explained.

The knowledge of uncertainty can be used for a number of applications, which will be detailed in the next chapter. In the context of autonomous control the most important application is quality assurance. By optimizing a lower quantile of the  $Q$ -value distribution, one can decrease the probability of obtaining a policy that will perform poor for the real MDP and thus increase reliability.

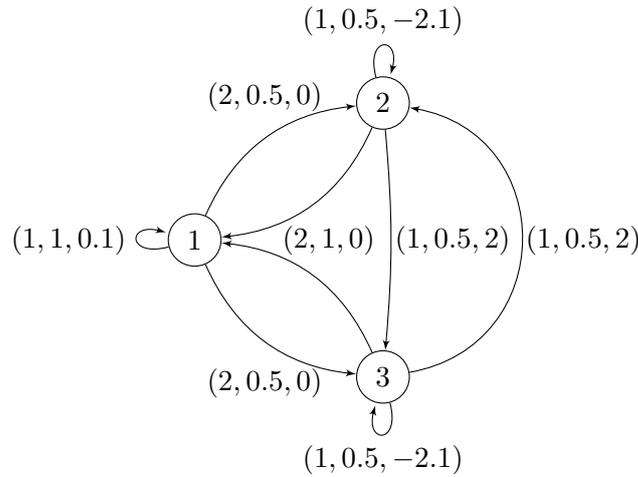
# 4

## Discrete Domain Applications of Uncertainty Awareness

As already outlined in the previous chapter, the knowledge of uncertainty can be used for various applications. This chapter will detail the applications and present a number of examples. First, the *quality assurance* application will be discussed, where instead of the expectation one optimizes a lower quantile of the  $Q$ -value distribution. While in general this leads to a lower expected performance, the probability of obtaining a poor solution is reduced. Although methods for discrete state and action spaces are not affected by most of the difficulties described in the introduction, the risk of having estimated the wrong Markov decision process (MDP) remains. For applications of autonomous control we are willing to accept a decreased expected performance for an increased minimal performance. In domains exhibiting the so-called *border phenomenon* even the expectation can be improved. Next, it will be discussed how to use the knowledge of uncertainty to enable *agent self-assessment*, i.e., evaluating a policy without executing it on an actual system. For the third and final application, we will change the sign of the parameter weighting the uncertainty and therefore obtain policies that seek state-action pairs with a high sum of current  $Q$ -value and corresponding uncertainty, thus realizing efficient exploration. Although exploration is not in the focus of this work, it shows the versatility of applications for uncertainty awareness.

### 4.1 Quality Assurance

Given the knowledge of an MDP, i.e., the state transition probabilities  $\mathcal{P}$  and the reward function  $\mathcal{R}$ , dynamic programming (DP), namely value iteration, can be used to determine the optimal  $Q$ -function  $Q^*$ , from which the optimal policy follows as  $\pi^*(s) := \arg \max_{a \in \mathcal{A}} Q^*(s, a)$ . If, however, only observations of the MDP in the form of  $(s, a, r, s')$  tuples, consisting of state  $s$ , action  $a$ , reward  $r$ , and successor state  $s'$ , are available, the MDP must be estimated from those observations before value iteration can be used. In that case, the resulting  $Q$ -function will be optimal w.r.t. the *estimated* MDP, and in general only sub-



**Figure 4.1:** Simple three-state MDP. In the description  $(a, b, c)$  of a transition  $a$  is the action,  $b$  the probability for that transition to occur, and  $c$  the reward.

optimal w.r.t. the real MDP.

One can use the knowledge of uncertainty to decrease the probability of obtaining a policy that will perform poor for the real MDP, thus determining policies of reasonable quality more reliably. For example, consider the simple three-state MDP in Figure 4.1. In state 1 the optimal policy would always execute action 1 in order to receive the certain reward of 0.1; in state 2 and state 3 the optimal policy would execute action 2 to go to state 1. If, however, the MDP's parameters had to be estimated from a set of observations and that set did contain an observation of the transition giving a reward of 2 from state 2 or state 3, but no observations of the transition giving a reward of  $-2.1$  in state 2 or state 3, the optimal policy for the estimated MDP would go to state 2 and execute action 1 there, expecting a reward of 2, while the true expected reward for that transition is  $0.5 \cdot (2 - 2.1) = -0.05$ . Obviously, this policy would perform far from optimal when applied to the real MDP. Not having observed the self-transition and corresponding reward of  $-2.1$  in either state 2 or state 3 even once, on the other hand, indicates a high uncertainty of the estimates for the transition probabilities from those state-action pairs. If one used one of the methods introduced in the previous chapter to determine the  $Q$ -function's uncertainty, it would become obvious that the  $Q$ -values of the actions chosen by the expectation-optimal policy are affected by high uncertainty. In a quality assurance setting we do not want a policy that is based on uncertain estimates. Instead, it should deliver a certain performance with high probability (i.e., little uncertainty), even if this performance is less than that of the expectation-optimal policy (whose performance, however, is affected by a higher uncertainty). To achieve that, in the policy update step of the value iteration, one does not choose the action that maximizes the  $Q$ -function, but

instead in the  $i$ -th iteration one sets

$$\forall s \in \mathcal{S} : \pi_i(s) \leftarrow \max_a Q_i(s, a) - \xi \sigma Q_i(s, a), \quad (4.1)$$

where  $\xi$  is a parameter weighting the uncertainty. The higher  $\xi$ , the more certain the return of the resulting policy will be. As described in the previous chapter, making assumptions about the distribution of the  $Q$ -values it is even possible to determine the probability of obtaining a return smaller than  $Q^\pi(s, a)$  when executing action  $a$  in state  $a$  and afterwards strictly following  $\pi$ , where  $\pi$  is determined according to (4.1).

### 4.1.1 Example

Consider the three-state MDP from Figure 4.1 again. Suppose further that we do not know the exact properties of the MDP; instead, we have made the observations shown in Table 4.1.

#	s	a	s'	r
1	1	1	1	0.1
2	1	1	1	0.1
3	1	1	1	0.1
4	1	2	2	0
5	2	2	1	0
6	1	2	3	0
7	3	1	2	2
8	3	1	2	2
9	3	1	2	2
10	2	1	3	2
11	3	1	3	-2.1
12	2	1	2	-2.1
13	3	2	1	0

**Table 4.1:** Exemplary observations of the simple three-state MDP.

Starting with a Dirichlet prior for each state-action pair with  $\alpha = 0.5$ , the observations were used to determine the posterior distributions. Using the expected values for  $\hat{\mathcal{P}}$  and  $\hat{\mathcal{R}}$ , the optimal  $Q$ -function was determined using value iteration; the  $Q$ -values and corresponding policy are shown in Table 4.2. With the given observations and the resulting estimates of  $\mathcal{P}$ , the policy tries to go from state 1 to state 2, then execute action 1 to go to state 3, and then go back to state 1 again. While this is indeed the optimal policy for the estimated MDP, it is not optimal for the real one. The only time this policy receives a reward other than zero is when executing action 1 in state 2. According to the estimated MDP,

the expected reward for that state-action pair is 2, while for the real MDP it is  $0.5 \cdot (-2.1 + 2) = -0.05$ .

state	Q(state, 1)	Q(state, 2)	$\pi$
1	2.53	2.65	2
2	2.63	2.51	1
3	3.36	2.51	1

**Table 4.2:**  $Q$ -values for the expectation-optimal policy according to the observations from Table 4.1.

The  $Q$ -values from Table 4.2 are those of the optimal policy w.r.t. the estimated MDP. To estimate the uncertainty of those  $Q$ -values, the sampling approach described in the previous chapter was used. From the Dirichlet posterior distribution 1,000 MDPs were sampled and on each sample policy evaluation was run for the policy determined previously. From that one can obtain an empirical distribution for each state-action pair. Table 4.3 shows the expected values and standard deviations from each  $Q$ -value distribution; Figure 4.2 gives the histograms (black). Looking at the values in Table 4.3, it becomes obvious that while the expected value of action 2 in state 1 is indeed higher, this advantage is not certain at all; when considering the uncertainties as well, both actions seem equally good. Likewise, the other  $Q$ -values exhibit high uncertainty, because they are influenced by the policy's choice of state-action pairs whose corresponding estimates of transition probabilities are affected by high uncertainty.

state	Q(state, 1)	Q(state, 2)
1	1.21 (5.28)	1.23 (5.71)
2	1.05 (6.76)	1.17 (5.41)
3	1.72 (6.18)	1.15 (5.42)

**Table 4.3:**  $Q$ -values and standard deviation (in brackets) estimated using the sampling approach based on 1,000 MDP samples.

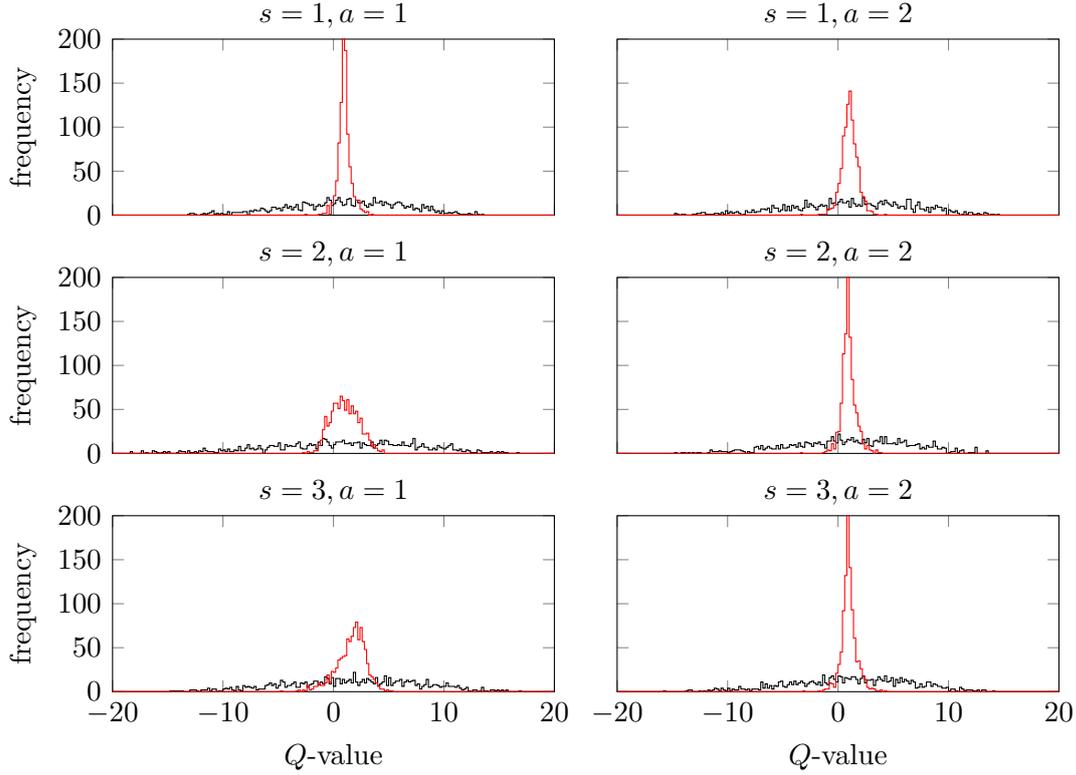
One might wonder about the difference between the  $Q$ -values in Tables 4.2 and 4.3. For the values in Table 4.2 the expected values from the Dirichlet distributions were used. For each state-action pair the multinomial distribution, i.e., the probabilities for successor states from this state-action pair, was assumed that is given by the expected value of the corresponding Dirichlet distribution. The set of multinomial distributions (one for each state-action pair) defines the MDP for which then policy evaluation was performed, leading to the values in the table. On the other hand, for the results in Table 4.3 sampled MDPs were used. For an MDP sample, for each state-action pair a multinomial distribution was drawn from the corresponding Dirichlet distribution. The resulting MDP sample was then used in policy evaluation (of the same policy), leading to one set of  $Q$ -value

samples. This was done 1,000 times, leading to 1,000  $Q$ -value samples for each state-action pair. From those samples the means and standard deviations were calculated, giving the values in the second table. One might expect that with an increasing number of MDP samples the values given by this Monte Carlo approach converge to those calculated for the expectation MDP, but this is not the case. The reason for this lies in the fact that  $\mathbf{E}_i f(x_i) \neq f(\mathbf{E}_i x_i)$  if  $f$  is a non-linear function. In our case,  $f$  is the Bellman iteration, i.e., the repeated application of the Bellman operator.  $x_i$  corresponds to the transition probabilities. In one case, the Bellman iteration is performed for the expected MDP (corresponding to  $f(\mathbf{E}_i x_i)$ ), in the other case the Bellman iteration is performed for MDP samples and the expectation is taken over the results (corresponding to  $\mathbf{E}_i f(x_i)$ ). The latter case is what we actually want, hence the Monte Carlo approach is the only one that correctly approximates the  $Q$ -value distribution. With the number of MDP samples going to infinity, the true distribution is revealed. Since for each MDP sample a complete Bellman iteration needs to be performed, the Monte Carlo approach is quite expensive. Therefore, in practice the full-matrix approach by Schneegaß, Udluft, and Martinetz (2008) as well as its approximation DUIPI are preferred, although both consider the expected MDP only and hence deliver approximate solutions. Nonetheless, both approaches work well in practice, as the experiments show.

Let us again consider the problem of a policy selecting an action whose superiority is uncertain and that in fact might be an inferior action. To circumvent this problem, the policy should select actions based not on the expected value, but a lower quantile. When assuming a normal distribution, selecting actions according to  $Q(s, a) - \xi\sigma Q(s, a)$  with, e.g.,  $\xi = 2$ , would yield a policy that with probability  $\Pr(\xi) = \Pr(2) \approx 0.977$  will perform at least as good as  $Q(s, a) - \xi\sigma Q(s, a)$  indicates.

Table 4.4 shows the  $Q$ -values and uncertainty of a policy determined using  $\pi(s) := \arg \max_{a \in \mathcal{A}} Q(s, a) - \xi\sigma Q(s, a)$ , with  $\xi = 1$ , in the iteration; the corresponding histograms are given in Figure 4.2. The resulting uncertainties are smaller, the distributions narrower. Although the uncertainties are still high, being uncertainty aware leads to a more “careful” policy here that stays in state 1 and receives the reward of 0.1 it is certain about, instead of trying to go for the uncertain 2 in state 2.

For the remainder of this chapter, we will move away from the sampling approach and instead use the methods that directly determine the uncertainty, namely the full-matrix uncertainty propagation (Schneegaß, Udluft, and Martinetz, 2008), DUIPI, and, for exploration, DUIPI-QM.



**Figure 4.2:** Histograms of  $Q$ -values of the three-state MDP. The black histograms correspond to the expectation-optimal policy, which chooses action 2 in state 1. Since the corresponding estimates are affected by high uncertainty, the resulting histograms are quite wide. The histograms of  $\pi^\xi$  with  $\xi = 1$  are shown in red. Since it considers both, the expected value as well as the corresponding estimate's uncertainty, the resulting histograms are narrower.

state	$Q(s, 1)$	$Q(s, 2)$	$(Q - \xi\sigma Q)(s, 1)$	$(Q - \xi\sigma Q)(s, 2)$	$\pi^\xi$
1	0.99 (0.56)	1.04 (0.72)	0.43	0.32	1
2	1.05 (1.26)	0.97 (0.66)	-0.21	0.31	2
3	1.60 (1.34)	1.00 (0.70)	0.26	0.30	2

**Table 4.4:**  $Q$ -values and standard deviation (in brackets) for  $\pi^\xi$ ,  $\xi = 1$ .

## 4.1.2 Benchmarks

In the quality assurance context experiments using the wet-chicken 2-D and archery benchmarks were performed.

### Wet-Chicken 2-D

Wet-chicken 2-D is a two-dimensional version of the original wet-chicken benchmark (Tresp, 1994).

In the original setting a canoeist paddles on a one-dimensional river with length  $l$  and flow velocity  $v = 1$ . At position  $x = l$  of the river there is a waterfall. Starting at position  $x = 0$  the canoeist has to try to get as near as possible to the waterfall without falling down. If he falls down, he has to restart at position  $x = 0$ . The reward increases linearly with the proximity to the waterfall and is given by  $r = x$ . The canoeist has the possibility to drift ( $x - 0 + v = x + 1$ ), to hold the position ( $x - 1 + v = x$ ), or to paddle back ( $x - 2 + v = x - 1$ ). River turbulence of size  $s = 2.5$  causes the state transitions to be stochastic. Thus, after having applied the canoeist's action to his position (also considering the flow of the river), the new position is finally given by  $x' = x + n$ , where  $n \in [-s, s]$  is a uniformly distributed random value.

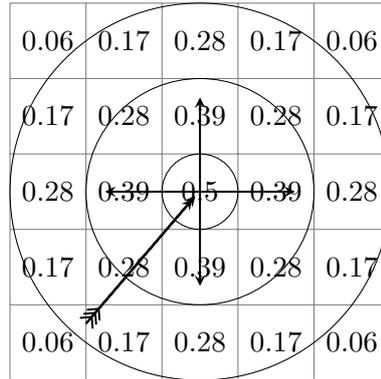
For the two-dimensional version the river is extended by a width  $w$ . Accordingly, there are two additional actions available to the canoeist, one to move the canoe to the left and one to move it to the right by one unit. The position of the canoeist is now denoted by  $(x, y)$ , the (re-)starting position is  $(0, 0)$ .

The velocity of the flow  $v$  and the amount of turbulence  $s$  depend on  $y$ :  $v = 3y/w$  and  $s = 3.5 - v$ . In the discrete problem setting, which we use here,  $x$  and  $y$  are always rounded to the next integer value.

While on the left edge of the river the flow velocity is zero, the amount of turbulence is maximal; on the right edge there is no turbulence (in the discrete setting), but the velocity is too high to paddle back.

### Archery

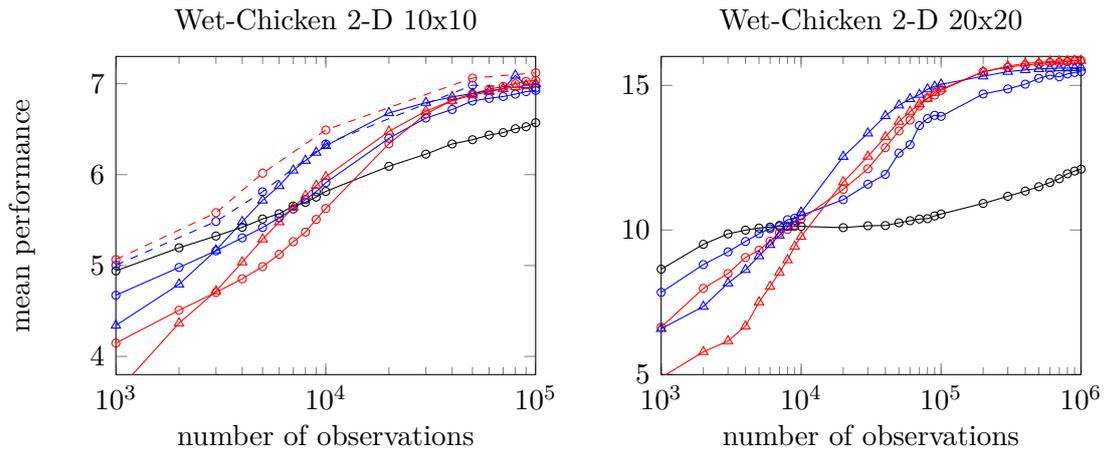
In the archery benchmark (Schneegaß, Udluft, and Martinetz, 2008), the state space represents an archer's target (Figure 4.3). Starting in the target's middle, the archer has the possibility to move the arrowhead in all four directions and to shoot the arrow. The exploration was performed randomly with short episodes of 25 transitions. The arrowhead's moves are stochastic (probability 0.25 of moving in another direction) as well as the event of making a hit after shooting the arrow. The highest probability for a hit is with the arrowhead in the target's middle. Every exploration episode starts in the middle as well. The border is explored quite rarely, such that a hit there can misleadingly cause the corresponding estimate to indicate a high reward and thus the agent to finally shoot from this place.



**Figure 4.3:** Visualization of the archery benchmark showing the 25 states with their hitting probabilities.

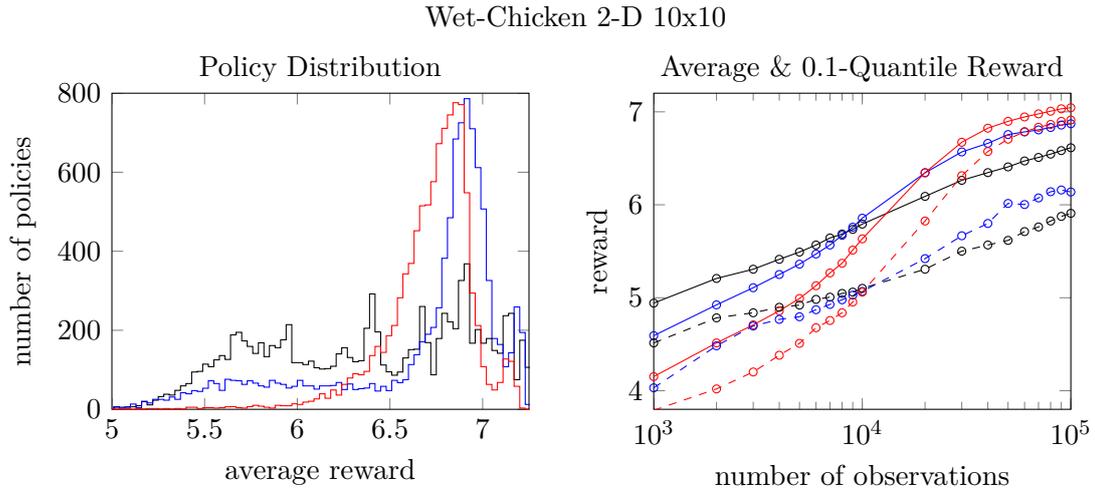
### 4.1.3 Experiments and Results

#### Wet-Chicken 2-D



**Figure 4.4:** Performance of policies generated using standard value iteration (black), DUIPI (solid lines), and the full-matrix method (dashed lines).  $\xi = 0.5$  is indicated by blue,  $\xi = 1$  by red. Policies generated using frequentist estimators are indicated by ‘ $\Delta$ ’ marks, ‘ $\circ$ ’ marks indicate policies generated using Bayesian estimation.

For the experiments river sizes of 10x10 (100 states) and 20x20 (400 states) were used. For both settings a fixed number of observations was generated using random exploration. The observations were used as input to generate policies using the different algorithms. The discount factor was chosen as  $\gamma = 0.95$ . Each resulting policy was evaluated over 100 episodes with 1,000 steps each. The results are summarized in Figure 4.4 (averaged over 100 trials). For clarity only the results of stochastic policies are shown (except for  $\xi = 0$ , i.e., standard policy iteration), they performed better than the deterministic ones in all experiments.



**Figure 4.5:** Left: histograms of average rewards of  $10^4$  policies with  $\xi = 0$  (solid),  $\xi = 1$  (blue), and  $\xi = 2$  (red). For the generation of each policy  $4 \cdot 10^4$  observations were used. Right: mean (solid) and 0.1-quantile (dashed) average rewards of policies with  $\xi = 0$  (black),  $\xi = 0.5$  (blue), and  $\xi = 1$  (red).

Usually, a method like DUIPI aims at quantile optimization, i.e., reducing the probability of generating very poor policies at the expense of a lower expected average reward. However, in some cases it is even possible to increase the expected performance, when the MDP exhibits states that are rarely visited but potentially result in a high reward. For wet-chicken, states near the waterfall have those characteristics. An uncertainty-unaware policy would try to reach those states if there are observations leading to the conclusion that the probability of falling down is low, which in fact is high. Schneegaß, Udluft, and Martinetz (2008) report this as the “border phenomenon”, which by our more general explanation is included. Due to this effect it is possible to increase the average performance using uncertainty aware methods for policy generation, which can be seen from the figure.

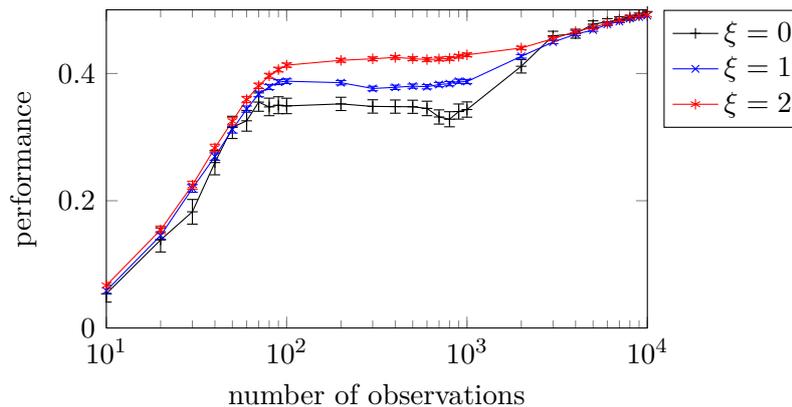
For small numbers of observations and high  $\xi$ -values DUIPI performs worse, as in those situations the action selection in the iteration is dominated by the uncertainty of the Q-values and not the Q-values themselves. This leads to a preference of actions with low uncertainty, the Q-values play only a minor role. This effect is increased by the fact that due to random exploration most observations are near the beginning of the river, where the immediate reward is low. Using a more intelligent exploration scheme could help to overcome this problem. Due to the large computational and memory requirements the full-matrix method could not be applied to the problem with river size 20x20.

Figure 4.5 compares uncertainty-aware and unaware methods. Considering the uncertainty reduces the amount of poor policies and even increases the expected performance ( $\xi = 0.5$ ). Setting  $\xi = 1$  results in an even lower probability for poor

policies at the expense of a lower expected average reward.

## Archery

In the archery domain the same experimental setup was used. For all experiments, the Bayesian estimator with  $\alpha = 0.001$  was applied. Figure 4.6 shows the mean performance, averaged over 1,000 trials. The standard approach not considering the uncertainty ( $\xi = 0$ ) starts similarly to the other approaches, with less than 100 observations it performs only slightly worse. Interestingly, however, when increasing the number of observations, the performance plateaus and even falls. This is because in the larger observation sets also more observations of border states are included. Since there are many border states, a hit by chance from one of those states is likely. Only when the number of observations is increased even further, the border states get explored more often as well and the estimates of the hitting probabilities of those state-action pairs get better. The uncertainty-aware approach, on the other hand, is not fooled by hits from border states. Since they are in fact unlikely, an estimate of a high hitting probability from a border state is necessarily affected by high uncertainty.



**Figure 4.6:** Results for the archery domain. Note that the x-axis has logarithmic scale.

## 4.2 Self-Assessment

When using reinforcement learning (RL) for optimal control of complex technical systems, often batch-mode RL is used, because many data-efficient RL methods operate in batch-mode and attain data-efficiency through re-use of observation tuples (Kalyanakrishnan and Stone, 2007). Furthermore, for many technical control tasks it is adequate to use batch-mode RL—observations of the system from operation with previous controllers are often available and the policy is not updated continuously, but only when a sufficient number of new observations leading to a substantially different policy are available.

When a new policy has been generated, it is usually advisable to evaluate it before applying it to the actual system to ensure its quality. In benchmark applications, this evaluation can easily be done using the environment itself. For a real-world application, however, this is usually not possible, as the policy might turn out to be insufficient, leading to an undesirable decrease of performance or even damage the system while being evaluated. As an alternative, one could use a simulation, but therefore such a simulation must be available and model the real system accurately enough to allow conclusions about the policy's performance on the real system.

Instead of actually executing the policy to evaluate it, we want to inspect a policy without requiring execution. The obvious indicator of policy performance is the value function. It should give the expected discounted future reward when following that policy. When the parameters of the MDP are known, policy evaluation can be used to determine the value function. However, if the knowledge of the environment is limited, the estimates of the underlying MDP's parameters might lead to wrong conclusions and thus a flawed value function that does not reflect the true performance of the policy on the *real* MDP.

### 4.2.1 Value Function-Based Self-Assessment

If the true value function of a policy is available, the obvious solution for self-assessment is the usage of the value function as indicator of policy quality. The expected return of a policy  $\pi$  is then given as

$$J^\mu(\pi) = \sum_{s \in S} \mu_0(s) V^\pi(s), \quad (4.2)$$

where  $\mu_0(s)$  is the probability of starting in  $s$  and  $V^\pi$  the value function of  $\pi$ . It showed that the mean value, i.e.,

$$J(\pi) = \bar{V}^\pi = \frac{1}{|S|} \sum_{s \in S} V^\pi(s), \quad (4.3)$$

is a good alternative; it was used for the experiments (Section 4.2.2). Given a set of policies, with  $J(\pi)$  it is possible to select the best  $m$  policies. Likewise, the user can specify a minimum required return  $J_{\min}$ . In an autonomous system a new policy  $\pi$  is then only applied if  $J(\pi) \geq J_{\min}$ .

Unfortunately, calculating the true value function requires exact knowledge of the MDP's state-transition probabilities and reward function. Usually, those are unknown and have to be estimated from observations. When dealing with stochastic MDPs, the estimates can be flawed, as was illustrated in Section 4.1. Having only a small set of observations (like the ones given in Table 4.1) to derive the estimates for the MDP and doing policy evaluation using those estimates, an inferior policy can have a higher value function than the optimal one and

consequently have a higher  $J(\pi)$  value. The problem here is that the estimates are used without considering their uncertainty. Knowing the uncertainty  $\sigma V^\pi$ , which stems from both, transition probability and reward uncertainties, one can reformulate equation (6.1) to

$$J_u^\mu(\pi) = \sum_{s \in \mathcal{S}} \mu_0(s) [V^\pi(s) - \xi \sigma V^\pi(s)], \quad (4.4)$$

where again  $\xi$  is a parameter weighting the uncertainty. Likewise, equation (4.3) becomes

$$J_u(\pi) = \frac{1}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} V^\pi(s) - \xi \sigma V^\pi(s). \quad (4.5)$$

The uncertainty incorporating value function  $V_u^{\pi, \xi}(s) = V^\pi(s) - \xi \sigma V^\pi(s)$  is called *quantile value function*.

## 4.2.2 Experiments

To evaluate the possibilities of determining policy quality using either the standard value function or the quantile value function, a number of experiments were conducted using the archery, wet-chicken 1-D (Tresp, 1994, see also Section 4.1.2), and trap (Dearden, Friedman, and Andre, 1999) benchmark domains.

### Setup

For each domain, a number of policies were generated. The aim was to select the best  $m$  policies without additional information, i.e., without running the policy on the real MDP or a simulation. Likewise, no additional observations were used. To assess the selection quality, each policy was evaluated on the real MDP; its performance (mean reward per step) served as a measure of its true quality.

In particular, for each experiment the following was done:

1. Generate  $N$  observations using random exploration, estimate the MDP from the observations, use dynamic programming (value iteration) to determine the optimal policy  $\pi_i$  for the estimated MDP, and finally evaluate the policy 100 times for 1,000 steps each to determine its performance  $\bar{r}_i$ . This is repeated 25 times, resulting in policies  $\pi_{1,2,\dots,25}$ , value functions and uncertainties  $(V, \sigma V)_{1,2,\dots,25}$ , and true performances  $\bar{r}_{1,2,\dots,25}$ .
2. Use  $J(\pi)$  and  $J_u(\pi)$  to create ranking vectors  $g^J$  and  $g^{J_u}$  of the policies. E.g.,  $g_1^J$  gives the index of the best policy according to  $J(\pi)$ ,  $g_{25}^J$  the worst.
3. From the true performances  $\bar{r}_i$  and the rankings  $g^J$  and  $g^{J_u}$  create vectors  $l^J$  and  $l^{J_u}$  containing the mean performance of the best  $m$  policies,  $m =$

$1, 2, \dots, 25$ , i.e.,  $l^J = \left( r_{g^J_1}, \frac{1}{2} \sum_{i=1}^2 \bar{r}_{g_i^J}, \dots, \frac{1}{m} \sum_{i=1}^m \bar{r}_{g_i^J}, \dots, \frac{1}{25} \sum_{i=1}^{25} \bar{r}_{g_i^J} \right)$ . Obviously,  $l_{25}^J = l_{25}^{J_u}$  gives the mean performance of all 25 policies.

4. Steps 1–3 are repeated 400 times, allowing to generate vectors  $\bar{l}^J$  and  $\bar{l}^{J_u}$  containing the mean of the individual  $l^J$  and  $l^{J_u}$  vectors and  $\sigma \bar{l}^J$  and  $\sigma \bar{l}^{J_u}$  containing the uncertainty of the mean (standard error).

Note that while the policies and their value functions and corresponding uncertainty were generated in one step, one could as well use a given policy and set of observations to generate the value function and uncertainty (policy evaluation).

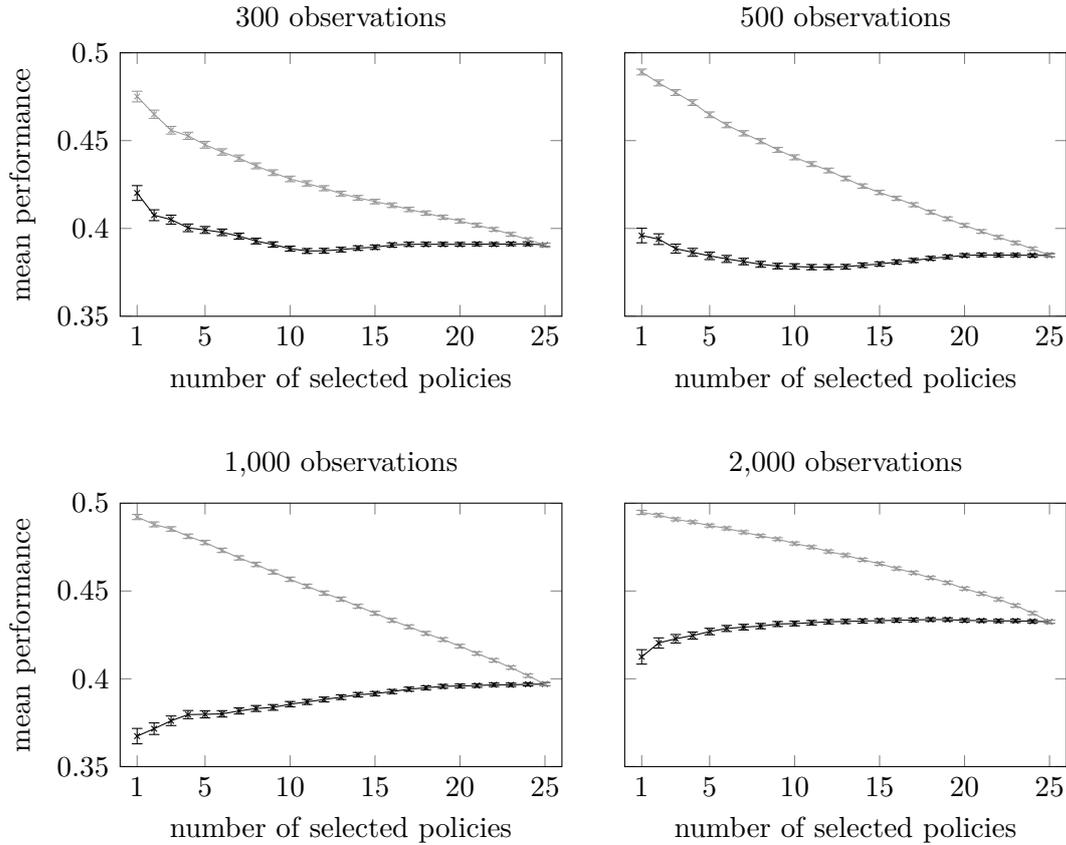
The discount factor was set  $\gamma = 0.975$ . For all experiments the Bayesian estimator with  $\alpha = 0.01$  for the transition probabilities was used; the sample mean served as estimate of the reward (Chapter 3, Section 3.2). As weighting of the uncertainty a value of  $\xi = 3$  was used.

The figures in this section show the mean rewards given a number of selected best policies (vectors  $\bar{l}^J$  and  $\bar{l}^{J_u}$ ). E.g., the very left point gives the mean performance of the best selected policy, the very right point gives the mean performance over all policies, i.e., the performance expectation when selecting a policy randomly. For each experiment, the 10,000 policies were divided into 400 distinct sets of 25 policies each, a ranking was performed for each of those sets of 25 policies. Therefore, each point in a figure is the average of 400 values.

## Archery

For the archery domain (Section 4.1.2), experiments according to the general setup were performed using various numbers of random exploration observations. Results for a representative selection of numbers of observations are given in Figure 4.7.

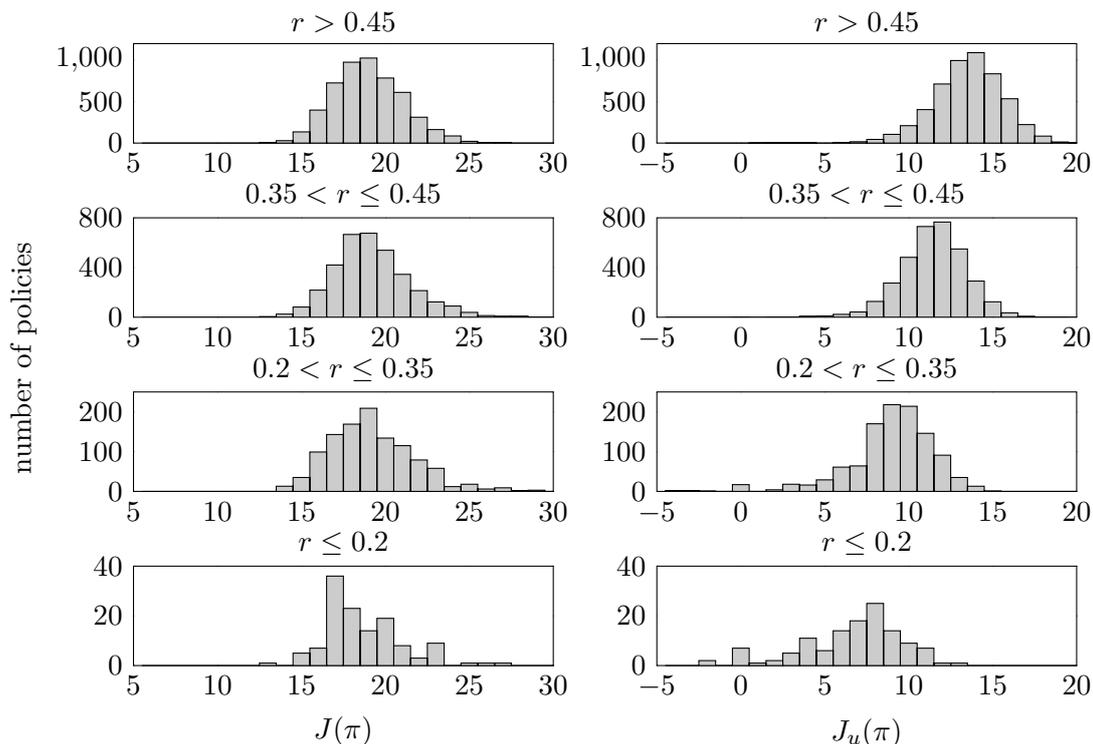
When estimating an MDP from 300 observations, the standard value function does help in selecting a policy performing better than average. When selecting only the presumably best policy from a set of 25 policies, the mean performance of the policies applied to the real problem is 0.43, while random selection gives policies with a mean performance of 0.39. However, when considering the uncertainty for the selection as well, the performance of the policies selected as best increases to 0.48. When using 500 observations, the gain achievable with standard value function based policy selection further decreases, while the selection quality of the quantile value function remains constant. This is because of the increasing number of misleading observations at the border of the target. Although the probability of hitting the target from a specific border state is quite small, since there are many border states, observing a hit from one of the border states is quite likely, leading to the assumption that shooting from this state the probability of hitting the target is high. This assumption leads to policies that move to such a border state and always shoot from there. With 1,000 and 2,000 observations,



**Figure 4.7:** Results of experiments using the archery benchmark. Shown are the performance of policies ranked either using  $J(\pi)$  (black) or  $J_u(\pi)$  (gray). The very left point in each plot shows the expected performance of the policy ranked best, the very right point gives the mean performance of all policies.

the problem becomes even more pronounced—selecting a policy based only on the standard value function leads to the selection of poor policies, as those are the ones with massively overestimated value functions. This is expected in domains exhibiting the “border phenomenon” (Schneegaß, Udluft, and Martinetz, 2008), where most observations are focused in a favorable area of the state space. The border is only explored rarely, but relatively large; it is therefore likely to observe a positive reward by chance. With increasing dimensionality of the state space, the border increases.

To illustrate the overestimation, Figure 4.8 shows histograms of the estimated mean values for different true policy qualities (exemplary for 2,000 observations). In the left column the standard value functions are depicted, the right column shows the histograms of the quantile values. The top row contains values for the best policies (mean reward greater than 0.4), the second and third row intermediate policies, the bottom row shows poor policies (mean reward less than 0.2). While the value function itself does not allow the selection of good policies



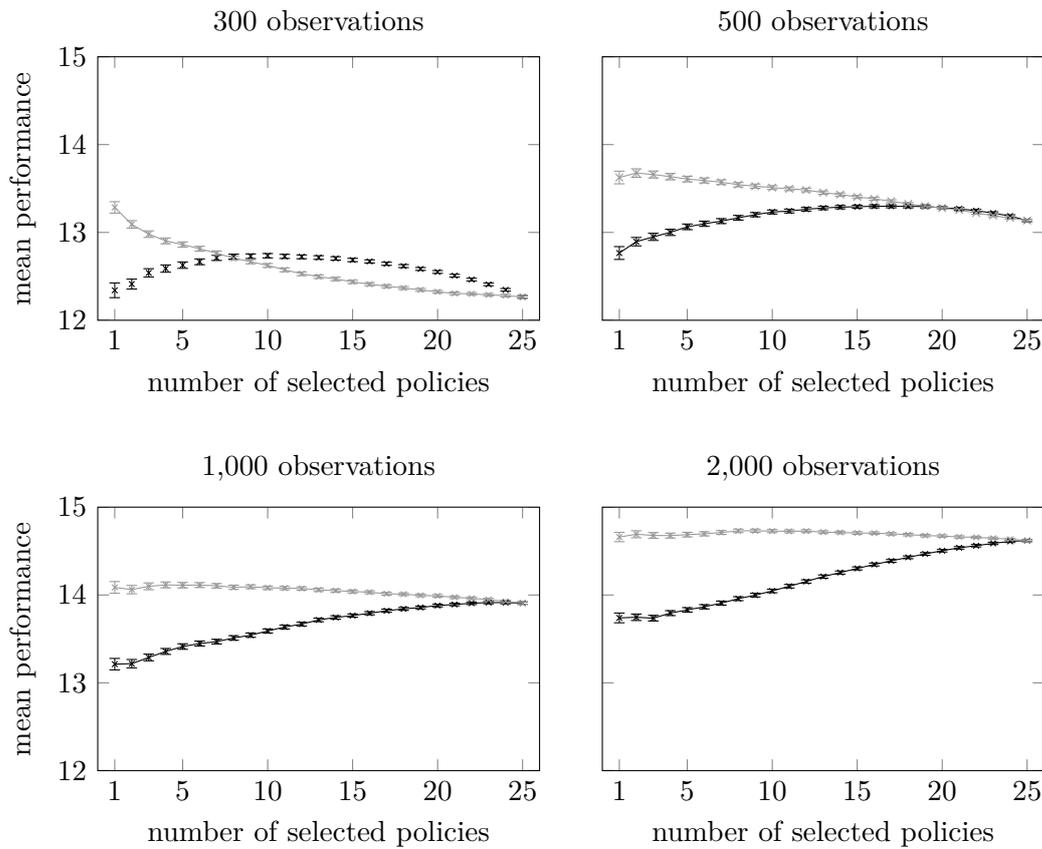
**Figure 4.8:** Histograms of  $J(\pi)$  and  $J_u(\pi)$  for different true policy performances for the experiment using the archery benchmark with 2,000 observations. The left column shows  $J(\pi)$  (ignoring uncertainty), the right column shows  $J_u(\pi)$  values considering the uncertainty. The policies are ordered by true performance with the best policies in the top row ( $r > 0.45$ ), intermediate policies in the second and third rows, and the worst policies in the fourth row ( $r \leq 0.2$ ).

(all histograms lie in the same range), the quantile value function reflects the true value more clearly (area of filled bins moves from lower to larger values with increasing true policy performance).

### Wet-Chicken

In the wet-chicken benchmark (Section 4.1.2), an exploration run consists of one continuous trajectory. Due to the stochasticity of the turbulences, there are situations when the canoeist is very near the waterfall without falling down. Although the probability of falling down from a point like this is high, limited observations can cause the estimator to misleadingly indicate a high probability of not falling down. Since the reward close to the waterfall is high, a policy generated using those estimates would try to reach a point near the waterfall, expecting to stay there without falling down and to receive a high reward.

Figure 4.9 shows the results for different numbers of observations. In the wet-chicken domain the policy selection using the value function systematically selects

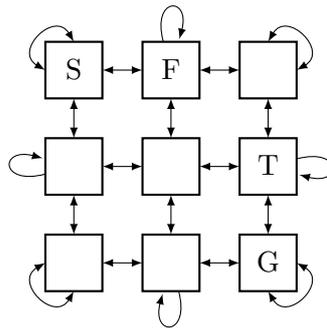


**Figure 4.9:** Performance of ranked policies for the wet-chicken benchmark. The ranking using the standard value function is marked black, the ranking according to the quantile value function gray.

poor policies. In this setting it is better to pick a policy randomly than choosing the one with the best value function. E.g., for 2,000 observations most policies are near-optimal, but the over-optimistic value function of some policies leads to the selection of poor policies. When considering the uncertainty (with the quantile value function), the situation changes, since the overestimated values are affected by a high uncertainty.

## Trap

The trap domain is a maze containing 18 states and four possible actions (Dear- den, Friedman, and Andre, 1999). The agent must collect flags and deliver them to the goal. For each flag delivered the agent receives a reward. However, the maze also contains a trap state. Entering the trap state results in a large negative reward. With probability 0.8 the agent’s action has the desired effect, with probability 0.2 the agent moves in perpendicular direction (chosen randomly with equal probability). See Figure 4.10 for an illustration.



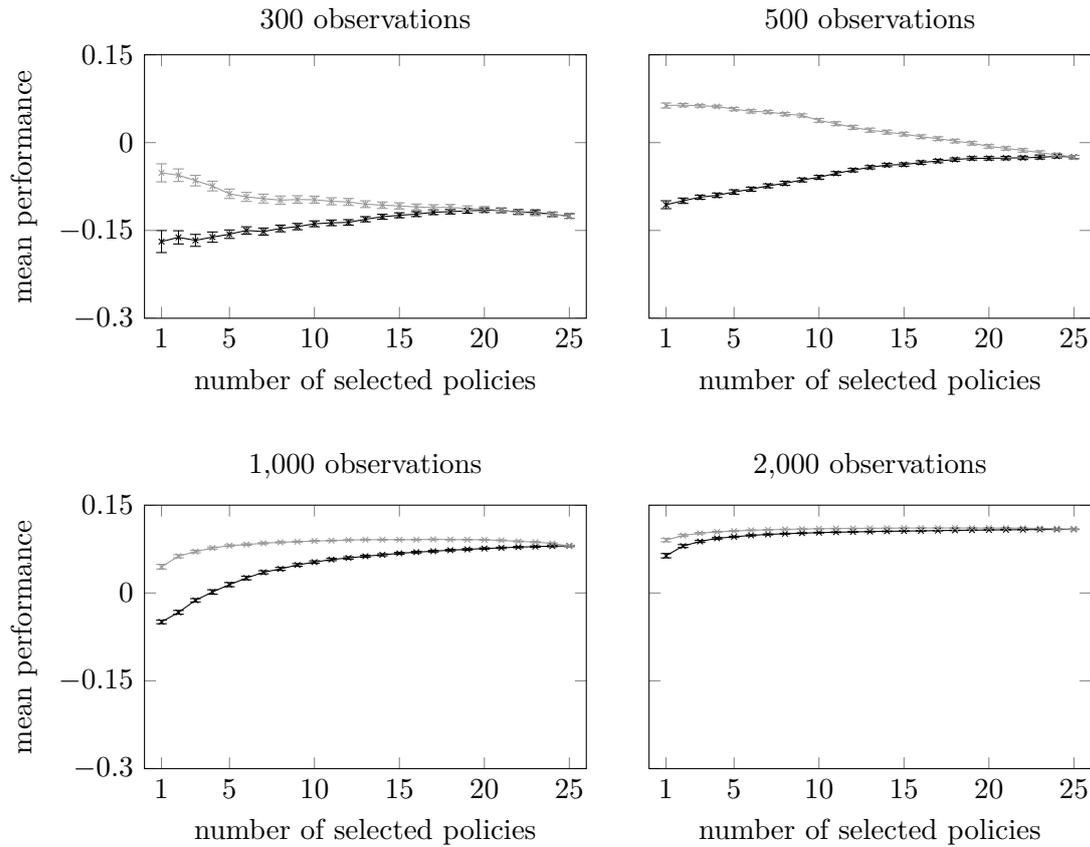
**Figure 4.10:** Illustration of the trap domain. Starting in state S the agent must collect the flag from state F and deliver it to the goal state G. Once the flag is delivered to state G, the agent receives a reward of 1 and is transferred to the start state S again. Upon entering the trap state T a large negative reward of  $-2.1$  is given. All other states yield a reward of 0. In each state the agent can move in all four directions. With probability 0.9 it moves in the desired direction, with probability 0.1 it moves in one of the perpendicular directions with equal probability.

The results from this domain are shown in Figure 4.11. For 300 and 500 observations we see the same effects as with the other domains—while the value function based approach systematically selects poor policies, considering the uncertainty of the value function as well it is possible to overcome this problem and select good policies. However, for 1,000 and 2,000 observations, in the setting chosen here with  $\xi = 3$ , also the uncertainty aware approach tends to systematically select poor policies, albeit not as extreme as the value function only approach. Setting  $\xi$  to a higher value would help here, but could lead to a dominance of the uncertainty for cases with fewer observations.

The optimal policy for this domain tries to stay away from the trap state. After collecting the flag, it goes back to the start state, then two fields down, and finally two fields right to deliver the flag. If an observation set does not contain the event of entering the trap state accidentally from the state left of it, the resulting policy will try to take the shortest path from the flag state to the goal, closely passing the trap state. Since the path is shorter, the corresponding estimated value function will misleadingly contain larger values than those of a more defensive (and in fact better) policy.

### 4.2.3 Related Work

To the best of our knowledge, so far only few works related to the issue of self-assessment in RL exist. There are works concerned with the selection of a suitable policy. Gabel and Riedmiller (2006) address the problem of policy degradation in NFQ by calculating a sample of the optimal  $Q$ -function tabularly and comparing that with the neural representation. They conclude that the closer the match, the



**Figure 4.11:** Performance of ranked policies for the trap domain. Again, the result of the standard value function based ranking is marked black, the ranking also incorporating the uncertainty is marked gray.

better the policy. Migliavacca et al. (2010, 2011) propose *fitted policy search*, a direct policy search method that uses an FQI-like approach to evaluate candidate policies. Instead of evaluating the policy on the real system or a simulation, they use *fitted policy evaluation* to determine the value function of the candidate policy using that policy and a set of observations of the system.

#### 4.2.4 Conclusion

The work in this section can be considered as a first attempt at comparing policies without executing them on the real MDP. It was shown that the value function can be misleading and largely overestimate the quality of the policy. To address this problem, uncertainty propagation was used to determine the uncertainty of the value function as well. Considering the uncertainty to determine the quantile value function, it becomes possible to much more reliably distinguish between good and poor policies. Although the discrete MDP setting considered here is not the preferred solution in practice—instead of estimating MDPs and corresponding

optimal policies from a number of distinct observation sets and then selecting from those policies, using all available observations to estimate a single MDP would usually yield better results—for continuous state (and action) MDPs it is an important issue. In Chapter 6 we will try to adapt the methods to continuous domains. Although it is not possible to do this in a straightforward manner, uncertainty and corresponding mis-estimation again play a central role.

So far we only dealt with methods that use an existing set of observations to derive or evaluate a policy, no matter where those observations came from and how they were generated. The next section will discuss how the knowledge of uncertainty can be used for efficient exploration.

## 4.3 Exploration

When no observations from previous interaction with the environment by some policy are available, one has to do exploration to learn about the environment. In an online setting, where each new observation is potentially used immediately to update the current policy, one is often interested in finding a good policy as quickly as possible and therefore explore efficiently, but at the same time the rewards gathered should be maximized right from the start. In that context the well-known exploration-exploitation dilemma arises: when should the agent stop trying to gain more information (explore) and start to act optimally w.r.t. already gathered information (exploit)?

Uncertainty awareness can also be used in this setting to combine existing (already gathered) knowledge and uncertainty about the environment to further explore areas that seem promising judging by the current knowledge. Moreover, by aiming at obtaining high rewards and decreasing uncertainty at the same time, good online performance is possible.

It will be shown that using a natural measure of the uncertainty obtained via uncertainty propagation (UP) it is possible to explore efficiently without relying on an artificial exploration bonus. Furthermore, in two variants of the algorithm the  $Q$ -function itself is not modified and still represents the followed policy and actually collected rewards. Moreover, no “optimistic” initialization of the  $Q$ -function is necessary.

### 4.3.1 Related Work

There have been many contributions considering efficient exploration in RL. In the following *Bayesian Q-learning* (Dearden, Friedman, and Russell, 1998), *model-based interval estimation* (Wiering and Schmidhuber, 1998; Strehl and Littman, 2009), and *R-Max* (Brafman and Tennenholtz, 2003) are considered.

### Bayesian Q-learning

Dearden, Friedman, and Russell (1998) presented *Bayesian Q-learning*, a Bayesian model-free approach that maintains probability distributions over  $Q$ -values. They either select an action stochastically according to the probability that it is optimal or select an action based on *value of information*, i.e., select the action that maximizes the sum of  $Q$ -value (according to the current belief) and expected gain in information. They later added a Bayesian model-based method that maintains a distribution over MDPs, determines value functions for sampled MDPs, and then uses those value functions to approximate the true value distribution (Dearden, Friedman, and Andre, 1999).

### Model-Based Interval Estimation

In *model-based interval estimation* (MBIE) (Wiering and Schmidhuber, 1998; Strehl and Littman, 2009) one tries to build confidence intervals for the transition probability and reward estimates and then optimistically selects the action maximizing the value within those confidence intervals. Strehl and Littman (2009) proved that MBIE is able to find near-optimal policies in polynomial time. This was first shown by Kearns and Singh (1998) for their  $E^3$  algorithm and later by Brafman and Tennenholtz (2003) for the simpler R-Max algorithm.

Strehl and Littman (2009) present an additional algorithm called *model-based interval estimation with exploration bonus* (MBIE-EB) of which they also prove optimality. According to their experiments, it performs similarly to MBIE. MBIE-EB alters the Bellman equation to include an exploration bonus term

$$Q(s, a) = \hat{\mathcal{R}}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \hat{\mathcal{P}}(s'|s, a) \max_{a' \in \mathcal{A}} Q(s', a') + \frac{\beta}{\sqrt{n_{s,a}}}, \quad (4.6)$$

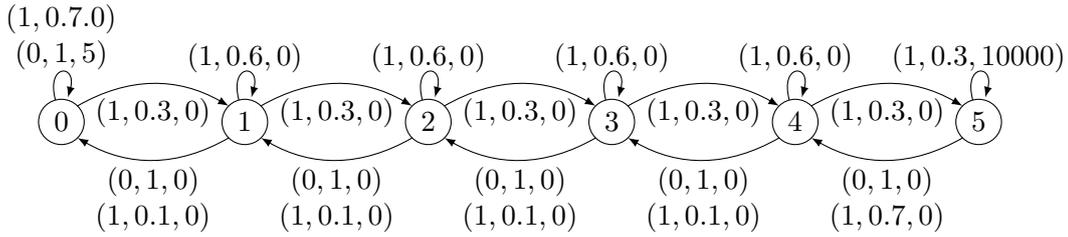
where  $\beta$  is a parameter of the algorithm and  $n_{s,a}$  the number of times state-action pair  $(s, a)$  has been observed.

### R-Max

R-Max takes a parameter  $C$ , which is the number of times a state-action pair  $(s, a)$  must have been observed until its actual  $Q$ -value estimate is used in the Bellman iteration. If it has been observed fewer times, its value is assumed as

$$Q(s, a) = \frac{r_{\max}}{1 - \gamma}, \quad (4.7)$$

which is the maximum possible  $Q$ -value ( $r_{\max}$  is the maximum possible reward). This way exploration of state-action pairs that have been observed fewer than  $C$  times is fostered.



**Figure 4.12:** Illustration of the river-swim domain. In the description  $(a, b, c)$  of a transition  $a$  is the action,  $b$  the probability for that transition to occur, and  $c$  the reward.

### Risk and Uncertainty

There have also been a number of contributions considering the incorporation of *risk* in RL (Heger, 1994; Neuneier and Mihatsch, 1998; Sato and Kobayashi, 2000; Geibel, 2001). These approaches deal with the risk of obtaining a low return in the single run—a risk that even exists for an optimal policy due to the inherent stochasticity of the MDP. The consideration of *uncertainty* deals with the uncertainty of the estimated parameters, due to our incomplete knowledge about the MDP. While this uncertainty decreases with an increasing number of observation, the stochasticity of the MDP and therefore the risk of obtaining a low return in the single run remains.

To demonstrate the functionality of the approach, experiments were conducted using two benchmark applications from the literature. The following section compares the full-matrix version, classic DUIPI, DUIPI with  $Q$ -function modification, and two established algorithms for exploration, R-Max (Brafman and Tennenholtz, 2003) and MBIE-EB (Strehl and Littman, 2009). Furthermore, some insight is presented on how the parameter  $\xi$  influences the agent’s behavior.

### 4.3.2 Benchmarks

The first benchmark is the *river-swim* domain from Strehl and Littman (2009), which is an MDP consisting of six states and two actions. The agent starts in one of the first two states (at the beginning of the row) and has the possibility to swim to the left (with the current) or to the right (against the current). While swimming to the left always succeeds, swimming to the right most often leaves the agent in the same state, sometimes leads to the state to the right, and occasionally (with small probability) even leads to the left. When swimming to the left in the very left state, the agent receives a small reward. When swimming to the right in the very right state, the agent receives a very large reward, for all other transitions the reward is zero. The optimal policy thus always swims to the right. See Figure 4.12 for an illustration.

The other domain considered here is the trap domain as used previously.

	river-swim	trap
<b>R-Max</b>	$3.02 \pm 0.03 \cdot 10^6$	$469 \pm 3$
<b>MBIE-EB</b>	$3.13 \pm 0.03 \cdot 10^6$	$558 \pm 3$
<b>full-matrix UP</b>	$2.59 \pm 0.08 \cdot 10^6$	$521 \pm 20$
<b>DUIPI</b>	$0.62 \pm 0.03 \cdot 10^6$	$554 \pm 10$
<b>DUIPI-QM</b>	$3.16 \pm 0.03 \cdot 10^6$	$565 \pm 11$

**Table 4.5:** Best results obtained using the various algorithms in the river-swim and trap domains. The used parameters can be found in Table 4.6.

	river-swim	trap
<b>R-Max</b>	$C = 16$	$C = 1$
<b>MBIE-EB</b>	$\beta = 0.01$	$\beta = 0.01$
<b>full-matrix UP</b>	$\alpha = 0.3, \xi = -1$	$\alpha = 0.3, \xi = -0.05$
<b>DUIPI</b>	$\alpha = 0.3, \xi = -2$	$\alpha = 0.1, \xi = -0.1$
<b>DUIPI-QM</b>	$\alpha = 0.3, \xi = -0.049$	$\alpha = 0.1, \xi = -0.049$

**Table 4.6:** Parameters used for the experiments.

For each experiment the cumulative reward for 5,000 steps was measured. The discount factor was set  $\gamma = 0.95$  for all experiments.

### 4.3.3 Results and Discussion

Table 4.5 shows the results for the considered domains and algorithms obtained with the respective parameters set to the optimal ones found. Reported are results averaged over multiple trials, for each average its uncertainty is given as well.

For river-swim, all algorithms except classic DUIPI perform comparably. By considering only the diagonal of the covariance matrix, DUIPI neglects the correlations between different state-action pairs. Those correlations are large for state-action pairs that have a significant probability of leading to the same successor state. In river-swim many state-action pairs have this property. Neglecting the correlations leads to an underestimation of the uncertainty, which prevents DUIPI from correctly propagating the uncertainty of  $Q$ -values of the right most state to states further left. Thus, although  $Q$ -values in state 5 have a large uncertainty throughout the run, the algorithm settles for exploiting the action in the left most state giving the small reward if it has not found the large reward after a few tries. DUIPI-QM does not suffer from this problem as it modifies  $Q$ -values using uncertainty. In DUIPI-QM, the uncertainty is propagated through the state space by means of the  $Q$ -values.

	full-matrix UP	DUIPI	DUIPI-QM
<b>time</b>	7 min	14 s	14 s

**Table 4.7:** Computation time for 5,000 steps in the river-swim domain using a single core of an Intel Core 2 Quad Q9500 processor. The policy was updated in every time step.

In the trap domain the correlations of different state-action pairs are less strong. As a consequence, DUIPI and DUIPI-QM perform equally well. Also the performance of MBIE-EB is good in this domain, only R-Max performs worse than the other algorithms. R-Max is the only algorithm that bases its explore/exploit decision solely on the number of executions of a specific state-action pair. Even with its parameter set to the lowest possible value, it often visits the trap state and spends more time exploring than the other algorithms.

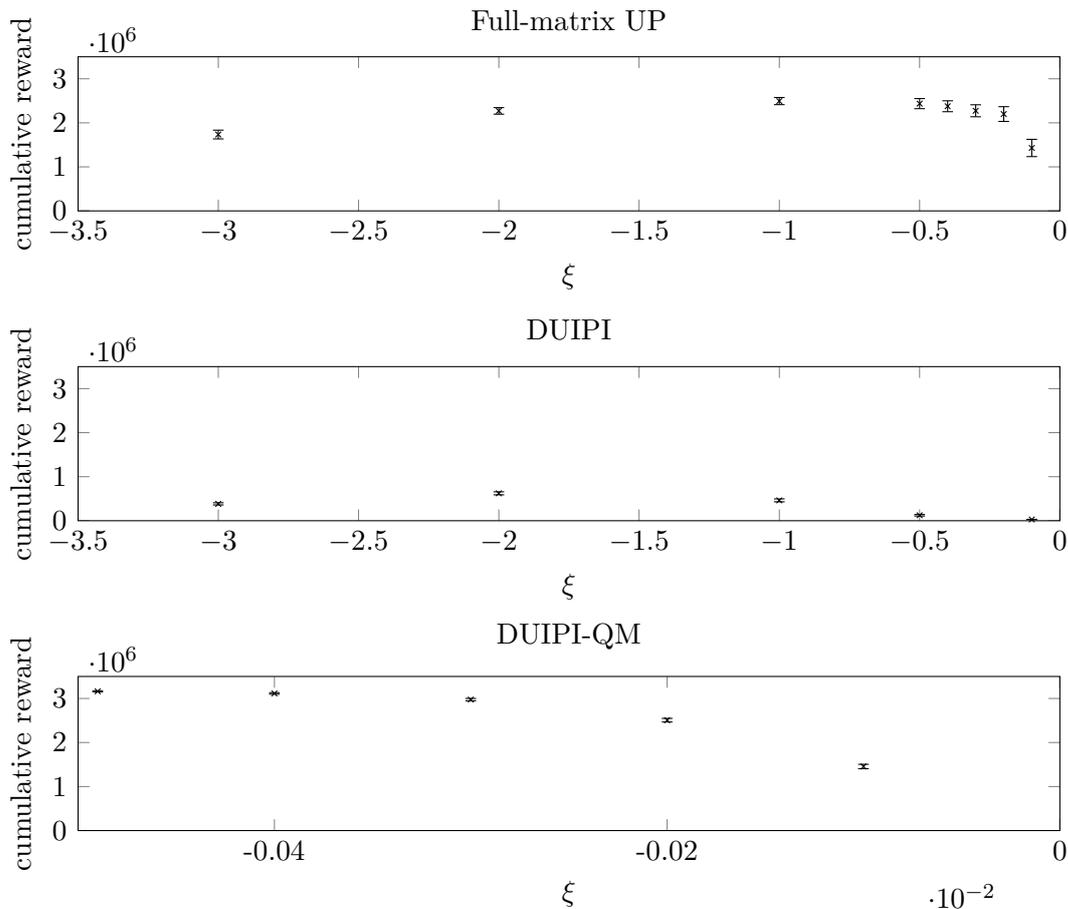
Although full-matrix UP performed worse than the approximate algorithm DUIPI-QM, it is in general expected to be the best performing algorithm; the author believes that the results here are due to peculiarities of the test domains.

Figure 4.13 shows the effect of  $\xi$  for the algorithms. Except DUIPI-QM the algorithms show “inverted u”-behavior. If  $\xi$  is too large (its absolute value too small), the agent does not explore much and quickly settles on a suboptimal policy. If, on the other hand,  $\xi$  is too small (its absolute value too large), the agent spends more time exploring. The author assumes that DUIPI-QM would exhibit the same behavior for smaller values for  $\xi$ , however, those are not usable, as they would lead to a divergence of  $Q$  and  $\sigma Q$ .

Figure 4.14 shows the effect of  $\xi$  using DUIPI in the trap domain. While with large  $\xi$  the agent quickly stops exploring the trap state and starts exploiting, with small  $\xi$  the uncertainty keeps the trap state attractive for more time steps, resulting in more negative rewards.

Using uncertainty as a natural incentive for exploration is achieved by applying uncertainty propagation to the Bellman equation. Our experiments indicate that it performs at least as good as established algorithms like R-Max and MBIE-EB. While most other approaches to exploration assume a specific statistical paradigm, our algorithm does not make such assumptions and can be combined with any estimator. Moreover, it does not rely on state-action pair counters, optimistic initialization of  $Q$ -values, or explicit exploration bonuses. Most importantly, when the user decides to stop exploration, the same method can be used to obtain quantile-optimal policies for quality assurance (Section 4.1) by setting  $\xi$  to a positive value.

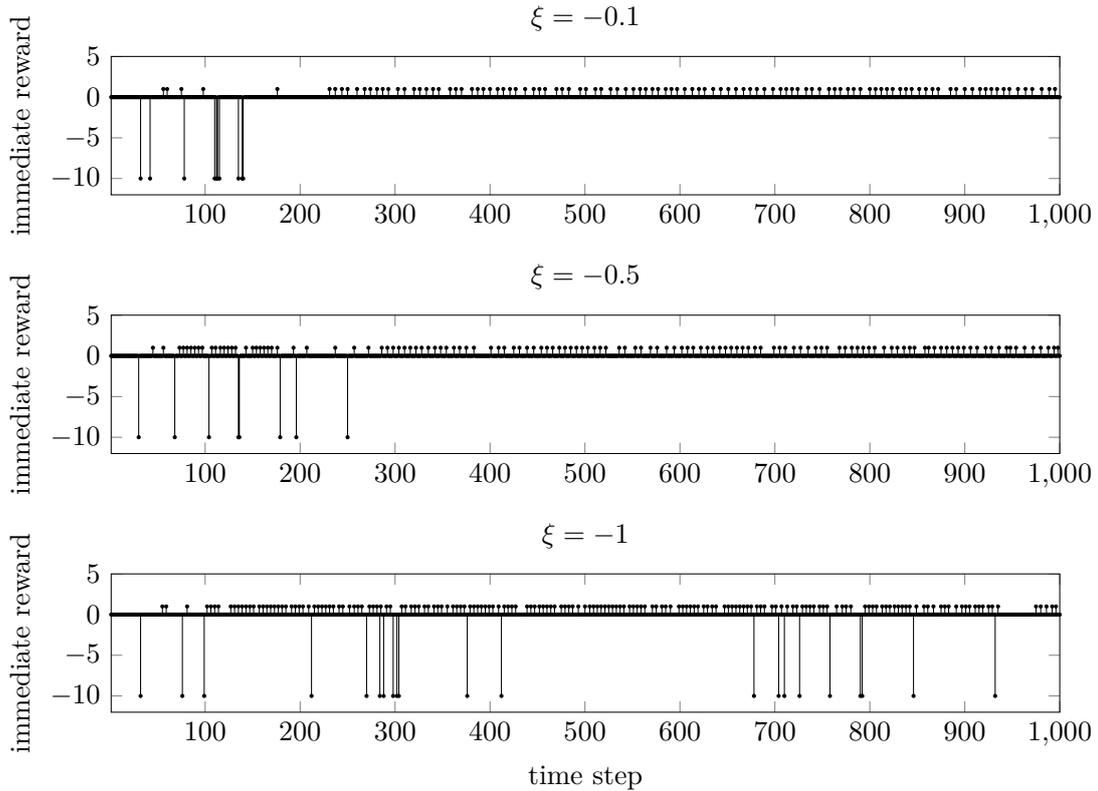
While full-matrix UP is the more fundamental and theoretically more sound method, its computational cost is considerable. If used with care, however, DUIPI and DUIPI-QM constitute valuable alternatives that proved well in practice.



**Figure 4.13:** Cumulative rewards for river-swim obtained by the algorithms for various values of  $\xi$ . The values for full-matrix UP are averaged over 50 trials, for the values for DUIPI and DUIPI-QM 1,000 trials of each experiment were performed.

## 4.4 Summary

This chapter concludes the part about methods for MDPs with discrete state and actions spaces. After having introduced ways of determining the  $Q$ -function's uncertainty and from that quantile-optimal policies in the previous chapter, corresponding applications were detailed here. We started with quality assurance, where one is interested in optimizing a lower quantile and thus decrease the probability of obtaining a poor policy. While this performance guarantee comes in general at the expense of a lower expected performance, the experiments showed that in domains exhibiting states that are visited rarely, but can potentially yield a high reward (though on average are poor and should be avoided), even the expected performance can be increased (“border phenomenon”). We started with a small three-state MDP to illustrate the effect of quantile optimality. It was further discussed why the Monte Carlo method arrives at other results than using the expected values of the Dirichlet distributions. Next, the full-matrix approach



**Figure 4.14:** Immediate rewards of exemplary runs using DUIPI in the trap domain. When delivering a flag, the agent receives reward 1, when entering the trap state it receives  $-2.1$ . While with  $\xi = -0.1$  after less than 300 steps the trap state does not seem worth exploring anymore, setting  $\xi = -0.5$  makes the agent explore longer due to uncertainty. With  $\xi = -1$  the agent does not stop exploring the trap state in the depicted 1,000 time steps.

as well as DUIPI were applied in a quality-assurance setting for the archery and wet-chicken 2-D benchmarks. Subsequently, self-assessment was discussed, i.e., assessment of a policy without requiring its execution on the actual MDP or additional observations. It was argued that the value function itself can be misleading. If also its uncertainties are considered, selection of good and poor policies becomes possible. This shows again the impact of mis-estimation and how uncertainty awareness can act as a remedy. Finally, uncertainty-aware methods were used for efficient exploration by negating the parameter  $\xi$  and thus using uncertainty as an added bonus. The introduced approaches were compared to established methods for efficient exploration, namely R-Max and MBIE-EB; it showed that they perform comparably. Advantages in this work’s uncertainty-propagation based methods are the fact that they do not assume a specific statistical paradigm and, most importantly, can be combined with the quality assurance approach by changing  $\xi$  to a positive value once the exploration is considered finished.



# 5

## Ensembles for More Reliable Policy Identification

Most problems of optimal control for technical systems feature a continuous state space. However, the methods introduced in the previous chapters are only applicable to problems with discrete state and actions spaces. To deal with the computational and storage problem that comes with the potentially infinite number of states of a continuous state space, it is inevitable to resort to some sort of function approximation. An extremely simple form of function approximation is a grid-like discretization of the state space. Each hypercube can then be treated as a discrete state and discrete methods become applicable. However, such a locally constant function approximation not only generalizes very poorly, it also violates the Markov property, because different states within a single hypercube are not distinguishable any more (the coarseness of the discretization influences the severity of the violation).

In this thesis, mainly neural networks are used as function approximators; they can deal with high-dimensional inputs and have excellent generalization capabilities (Hastie, Tibshirani, and Friedman, 2001, p.351). In particular, neural networks are employed in the context of neural fitted  $Q$ -iteration (NFQ).

The following section will first detail NFQ and describe the algorithm it builds on, *fitted  $Q$ -iteration*. We will then discuss the problems of NFQ that hinder its application to autonomous control. As a way of increasing the reliability of NFQ, the usage of ensembles is proposed. In that context, a number of ensemble methods will be named and evaluated on different benchmark problems.

### 5.1 Neural Fitted $Q$ -Iteration

*Neural fitted  $Q$ -iteration* (NFQ) (Riedmiller, 2005) is an instance of the *fitted  $Q$ -iteration* (FQI) algorithm. FQI was first introduced by Ernst, Geurts, and Wehenkel (2003) as a data-efficient batch-mode reinforcement learning (RL) method (see also Ernst, Geurts, and Wehenkel, 2005). They used *extremely randomized trees* (Geurts, Ernst, and Wehenkel, 2006) as function approximator. In NFQ,

instead of regression trees a neural network is used.

### 5.1.1 Fitted Q-Iteration

Algorithm 6 summarizes FQI. It can be considered as a sample-based version of value iteration (see Chapter 2, Section 2.4.2). Instead of the transition probabilities, one uses the samples directly. Starting with an arbitrary  $Q$ -function (lines 2–3, assuming  $Q_{\text{init}} = 0$  here), a function approximator is trained to map inputs to  $Q$ -values (line 6). In the next step, the new targets are determined using an update based on the Bellman optimality equation (line 7). Since a (generalizing) function approximator was trained in line 6, it is possible to determine the  $Q$ -values for state-action pairs that are not contained in the set of observations  $\mathcal{O}$  and thus to maximize the  $Q$ -function over  $a'$ .

---

#### Algorithm 6: Fitted Q-Iteration

---

**Input:** set of observations  $\mathcal{O} = \{(s_i, a_i, r_i, s'_i) | i = 1, \dots, M\}$ ,  $\gamma \in [0, 1)$

**Result:** near-optimal  $Q$ -function  $Q^*$

```

1 begin
   | inputs are state-action pairs from observations
2   inputi := (si, ai)  ∀i ∈ {1, ..., M}
   | set rewards as initial targets
3   targeti := ri  ∀i ∈ {1, ..., M}
4   k := 0
5   while stopping criteria not reached do
   |   train regression algorithm to map input ↦ target
6   |   Qk := train(input, target)
   |   determine new targets
7   |   targeti := ri + γ maxa' Qk(s'i, a')  ∀i ∈ {1, ..., M}
8   |   k := k + 1
9   return Qk

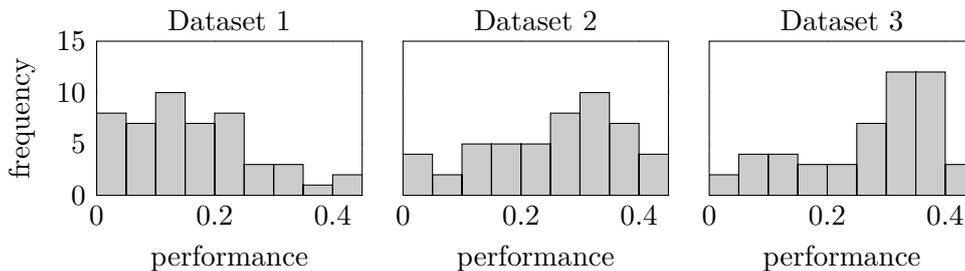
```

---

NFQ has proven to be very data-efficient. E.g., Riedmiller (2005) was able to learn near-optimal policies for the pole-balancing benchmark using observations from 100 episodes of random exploration (approximately 600 observations). However, a number of problems have been reported as well. In particular, it is hard to decide when a  $Q$ -function leading to a good policy is reached and therefore when to stop iterating.

### 5.1.2 Problems of NFQ

There have been a number of reports on problems regarding the learning process with function approximators in RL and FQI in general and NFQ in par-



**Figure 5.1:** Performances of policies from repeated NFQ runs on different datasets. With each of the three datasets NFQ was run 50 times, resulting in 50 policies. The histograms show the performances, i.e., average reward, of the policies for each dataset.

ticular (Thrun and Schwartz, 1993; Gordon, 2001; Gaskett, 2002; Gabel and Riedmiller, 2006).

As an example, consider the histograms shown in Figure 5.1. Here, three distinct datasets were used to run NFQ 50 times on each. This resulted in 50 policies for each dataset. The policies were evaluated (by running them on the actual environment) and the achieved average reward was determined as a measure of a policy’s performance. Each histogram contains therefore 50 values. It shows that repeated runs of NFQ on the same dataset can lead to quite different results. Although the distributions are different (dataset 1 has a tendency towards poor policies, whereas most policies generated from dataset 2 and especially dataset 3 are comparatively good), all histograms contain a fair amount of both, poor as well as good policies. The stochastic components of neural network training, namely the random initialization of weights and the stochastic pattern selection, make different results given the same dataset possible. Ideally, NFQ would only produce good policies. In the following, reasons for its failure are discussed.

### Chattering

When using function approximation for RL, a phenomenon called *chattering* can occur (Gordon, 2001). The space of possible  $Q$ -functions for an MDP representable by a function approximator contains so-called *greedy regions*. In such a region the policy resulting from greedy exploitation of the respective  $Q$ -function, i.e., following  $\pi(s) = \arg \max_a Q(s, a)$ , does not change. Each greedy region contains a *greedy point*, during the process of learning the  $Q$ -function represented by the function approximator moves to that point. If the greedy point lies within the greedy region, no problems arise. However, if the greedy point lies on the border of another or even outside the greedy region, an oscillation can occur with the  $Q$ -function moving from one greedy region to another. Gabel and Riedmiller (2006) report this problem for NFQ. The authors suggest doing a policy selection by monitoring the current policy’s quality and stopping the learning process once the quality declines. Their method works by calculating a sample of the

optimal  $Q$ -function tabularly and comparing the ranking of actions of the tabular  $Q$ -function with the neural one. They conclude that the closer the match, the better the neural  $Q$ -function. While the approach is indeed able to stabilize the learning process and produce high-quality policies more reliably, its major drawbacks are the limitation to discrete state spaces and the necessity of having observed multiple actions in the same state, since the tabular  $Q$ -function cannot generalize over actions.

### Rising Q Problem

Thrun and Schwartz (1993) report the problem of overestimation of  $Q$ -values, a fundamental flaw of value function based RL with function approximation. When learning with noisy data, the output of a function approximator will also be affected by noise. Although the noise has a mean of zero, an algorithm relying on the maximization of  $Q$ -values (this includes  $Q$ -learning and FQI-like algorithms) will systematically overestimate the  $Q$ -value, as it selects the maximum  $Q$ -value over all actions when determining the targets for the next learning step. Thus the noise is maximized as well. More formally, taking the maximum over several noisy values is an overestimation, since

$$\mathbf{E}\{\max \vec{a}\} \geq \max \mathbf{E}\{\vec{a}\}, \quad (5.1)$$

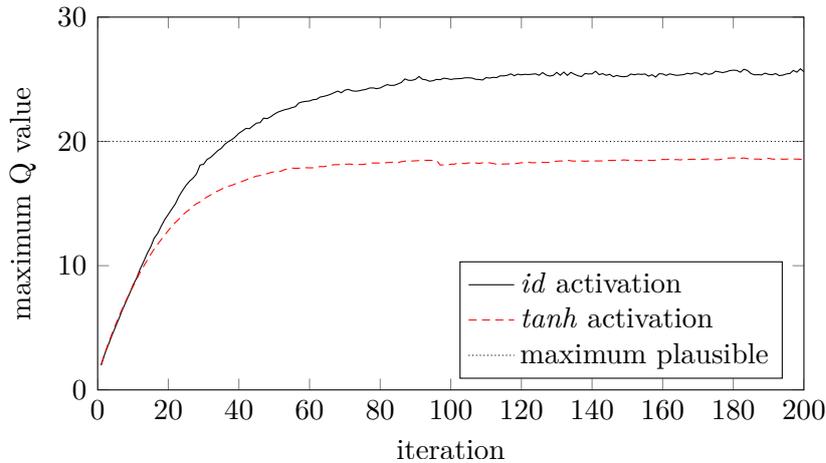
where  $\vec{a}$  is the vector of noisy values. Applied to the maximization of the  $Q$ -function, we have

$$\mathbf{E}\{\max_{a \in \mathcal{A}} Q(s, a)\} \geq \max_{a \in \mathcal{A}} \mathbf{E}\{Q(s, a)\}. \quad (5.2)$$

Gaskett (2002) observed that problem as well and called it the “rising Q problem”.

Van Hasselt (2010b) analyzed the overestimation of  $Q$ -values when applying tabular  $Q$ -learning in stochastic domains, without using function approximation. The stochasticity of the environment can have similar effects as the noisy output of a function approximator. Van Hasselt proposes *double Q-learning*, a variant of  $Q$ -learning where one uses two representations of the  $Q$ -function; when a  $Q$ -value of one representation is to be updated, the maximum successor state  $Q$ -value is determined using the other representation. He argues that the data-efficiency does not suffer, as the two representations can be combined to derive the policy. However, it is unclear whether this also applies in a setting with function approximation.

To weaken the rising  $Q$  problem, a sigmoid activation function is used for the output neuron of the  $Q$  network, thus bounding the range of possible output values. Using a bounded activation function for the output neuron was also reported to lead to a more stable learning behavior (Riedmiller, 2011), which is confirmed by the experiments here. The hyperbolic tangent serves as activation function, limiting the network’s outputs in  $[-1, 1]$ . To be able to represent  $Q$ -



**Figure 5.2:** Maximum  $Q$ -values per NFQ iteration with  $id$  and  $tanh$  activation functions in the output layer.

values smaller than  $-1$  or greater than  $1$ , a linear scaling is used ( $t_j$  denotes original targets,  $\hat{t}_j$  the scaled ones):

$$\kappa := 1.05 \max_j |t_j| \quad (5.3)$$

$$\hat{t}_i := \frac{t_i}{\kappa} \quad (5.4)$$

This way the maximum absolute value the network is asked to represent is  $|\hat{t}_{\max}| = 1/1.05 \approx 0.9524$ . Not having to use the complete range of possible outputs of  $[-1, 1]$  keeps the network from having to use large weights for the connections to the output layer. Obviously, the scaling value  $\kappa$  must be kept to be able to do an inverse scaling when network output values are to be used. Algorithm 7 summarizes the implementation. Gabel, Lutz, and Riedmiller (2011) independently proposed a scaling method similar to the one used here. The *train-net* function takes pairs of input vectors and corresponding target values, initializes a new neural network, and adapts the weights of the network to minimize the mean squared error (MSE) between the network’s output and the target values. To train a network, the *VarioEta* learning algorithm (Neuneier and Zimmermann, 1998) is employed. It leads to similar results as standard backpropagation (with small batches), but is often faster. The training is done in a number of steps, starting with a large learning rate and decreasing it after each step. In each step, the network is trained for a number of epochs  $n_e$ . If the average MSE of the last  $n_e$  epochs on a validation set did not fall by at least some pre-defined threshold, training in that step is stopped and the next step commences; otherwise, the network is trained for another  $n_e$  epochs.

To show the superiority of bounded output neurons in combination with scaling, a number of experiments were performed using the pole-balancing benchmark

	id noscale	id	tanh
<b>steps balanced</b>	284 (849)	2427 (1035)	2675 (866)
<b>successful policies</b>	3/50	23/50	37/50

**Table 5.1:** Number of successful policies (i.e., able to balance at least 3,000 steps) generated by NFQ runs with different activation functions and with and without target scaling (pole-balancing).

with fixed observation sets (see Section 5.4 for details on the benchmark and the data generation). With each data set, NFQ was run using different activation functions in the output layer. Each experiment was repeated 50 times. Figure 5.2 shows maximum  $Q$ -values for 200 iterations. In one case, the output neurons were unconstrained by using the identity as activation function. In the other case, the activation function was the hyperbolic tangent, limiting the network’s outputs in  $[-1, 1]$ . The dashed line denotes the maximum plausible  $Q$ -value  $Q_{\max} = R_{\max}/(1-\gamma)$ , with  $R_{\max}$  the maximum reward (in this case  $R_{\max} = 1$ ). As can be seen from the figure, when using the identity, the  $Q$ -values rise quickly and cross the line of the maximum plausible  $Q$ -value, indicating a clear overestimation. While the network using the hyperbolic tangent is able to overestimate as well (because of the 1.05 factor), it is harder, and the  $Q$ -value overestimation is not as severe as with the identity function. In addition to producing more plausible  $Q$ -values, the bounded output approach also leads to better policies. Table 5.1 shows the results. Additionally, the table contains results of the same experiment with *id* activation and no target scaling. It is obvious that the network’s ability to learn the  $Q$ -function and thus the quality of the resulting policy benefits from a target scaling, which is a common technique to successfully train neural networks (see, for instance, Bishop, 1995, Chapter 8).

### General Issues of Neural Network Training

Dietterich (2000) names three general problems that arise in the context of function approximation.

1. The statistical problem. With a limited number of training patterns, a solution that nicely fits both training and validation sets can still deviate from the true function.
2. The computational problem. Many algorithms, including neural networks, optimize a non-convex error function that exhibits local minima; by starting from different points in the parameter space and randomly selecting patterns for training, even given the same training data different instances of the same algorithms can arrive at different solutions.
3. The representational problem. The function approximator might even be

**Algorithm 7:** Scaling NFQ**Input:** set of observations  $\mathcal{O} = \{(s_i, a_i, r_i, s'_i | i = 1, \dots, N)\}$ ,  $\gamma$ ,  $K$ **Result:** near-optimal  $Q$ -function  $Q^*$ 


---

```

1 begin
   | inputs are state-action pairs from observations
2   inputi := (si, ai)  ∀i ∈ {1, ..., N}
   | set rewards as initial targets
3   targeti := ri  ∀i ∈ {1, ..., N}
4   k := 0
5   while k < K do
   |   scale targets
6   |   κk := 1.05 maxi |targeti|
7   |    $\widehat{\text{target}}_i := \frac{\text{target}_i}{\kappa_k} \quad \forall i \in \{1, \dots, N\}$ 
   |   train neural network
8   |   netQk := train-net(input,  $\widehat{\text{target}}$ )
   |   get maximum Q-values for successor states from network ...
9   |   Vk(s'i) := κk maxa' netQk(s'i, a')  ∀i ∈ {1, ..., N}
   |   ... and use them to determine new targets
10  |   targeti := ri + γVk(s'i)  ∀i ∈ {1, ..., N}
11  |   k := k + 1
12 return Qk

```

---

unable to exactly represent the actual function. Note that this problem can arise even if the function approximator itself is powerful enough (like a sufficiently large neural network), because the hypothesis space is limited by the size of the training set.

In addition to chattering and the rising  $Q$  problem, those general problems contribute to the instability of NFQ as well.

For the pole-balancing benchmark even with approximately 60,000 observations (10,000 episodes), a number that here excludes the statistical problem, the author could observe oscillations. While the vast majority of iterations produced successful policies, occasionally there was a policy that was unable to balance the pole for the required 3,000 steps. Therefore, at least in this case, adding observations alone is not sufficient.

To mitigate those problems, it is proposed to use ensembles for more robust and reliable RL with function approximation. This also makes the algorithm less sensitive to the possible choices of parameters.

## 5.2 Ensemble Methods

In supervised learning, ensemble methods such as mixture of experts (Jacobs, Jordan, Nowlan, and Hinton, 1991), bagging (Breiman, 1996), or boosting (Freund, Schapire, and Abe, 1999) have been used successfully to improve the performance of a single learner by combining several ones—both for classification and regression problems.

For classification problems, the most straightforward way of combining single learners to an ensemble is majority voting. Each individual classifier is trained independently on the dataset or a subset thereof. For new input data, each classifier votes for a class, the most voted for class is used as the final prediction of the ensemble. Ensembles improve the prediction quality if the individual members are *accurate* and *diverse* (Hansen and Salamon, 1990). A classifier is accurate if its error rate on new inputs is better than random guessing; the ensemble members are diverse if they do not make the same errors on new data points. For example, consider three classifiers  $\{h_1, h_2, h_3\}$ , each having an error rate of 0.4. If they are identical, i.e., not diverse, they will make the same errors on new inputs, their errors will be maximally correlated, and the ensemble error rate will consequently be identical to the individual ones. If, on the other hand, their errors are uncorrelated, i.e., the diversity is maximal, when  $h_1$  is wrong,  $h_2$  and  $h_3$  may be correct, so that the majority vote will give the correct result. Given  $M$  classifiers with error rates smaller than 0.5, more than  $M/2$  have to be wrong for the final classification to be wrong (Dietterich, 2000). One can see that in this case simply adding members to the ensemble will increase the prediction quality; with  $M \rightarrow \infty$ , the probability of more than  $M/2$  members being wrong tends to zero. In practice, however, the errors are not completely independent, therefore adding members makes only sense up to a certain point.

When using ensembles for regression problems, a simple aggregation scheme is the weighted mean of the outputs of the ensemble members as final output. In the context of neural network ensembles, Krogh and Vedelsby (1995) introduced the term *ambiguity* to quantify the disagreement of the members on a given input. Let  $f_i(x)$  be the output of network  $i$  on the input  $x$ . The ensemble output is then given by

$$\bar{f}(x) = \sum_{i=1}^M w_i f_i(x), \quad (5.5)$$

where  $w_i$  is the weight of network  $i$  and  $\sum_{i=1}^M w_i = 1$ . If no information about the quality of a specific network is available, all weights are set  $w_i = 1/M$ , i.e., every network is assigned the same weight. The ambiguity of a network  $i$  for an input  $x$  is the squared difference between the network's output  $f_i(x)$  and the ensemble output  $\bar{f}(x)$ , i.e.,

$$a_i(x) = (f_i(x) - \bar{f}(x))^2. \quad (5.6)$$

The ensemble ambiguity is the weighted sum of the ambiguities of the members:

$$\bar{a}(x) = \sum_{i=1}^M w_i a_i(x). \quad (5.7)$$

Based on this, one can express the average ensemble ambiguity over the input distribution  $X_0$  as

$$\bar{A} = \mathbf{E}_{x \sim X_0} \bar{a}(x). \quad (5.8)$$

Given the expected error of a single network of the ensemble as

$$\bar{E} = \frac{1}{M} \sum_{i=1}^M \mathbf{E}_{x \sim X_0} (f_i(x) - y(x))^2, \quad (5.9)$$

Krogh and Vedelsby (1995) showed that the ensemble error  $E$  can be expressed as

$$E = \bar{E} - \bar{A}, \quad (5.10)$$

i.e., the ensemble error is reduced by the ambiguity. This is in fact similar to the traditional bias/variance trade-off (Geman, Bienenstock, and Doursat, 1992). If the networks are strongly biased, they will all produce similar solutions, the ambiguity will be low and  $E \approx \bar{E}$ . If, on the other hand, they are less biased, the ambiguity will be higher and the ensemble error will be lower than the expected error of a single network (Krogh and Vedelsby, 1995; Brown, Wyatt, Harris, and Yao, 2005).

As we have seen, in an ensemble diversity is key. Consequently, ensemble methods like *bagging* (Breiman, 1996) or *boosting* (Freund, Schapire, and Abe, 1999) try to increase the diversity.

Bagging (short for *bootstrap aggregating*) uses bootstrapping (Efron, 1979; Efron and Tibshirani, 1993) to produce new datasets of the same size as the original; on each dataset, an individual learner is trained (Breiman, 1996). Bootstrapped replica are generated by repeatedly sampling with replacement from the original dataset. Given  $N$  samples in the original dataset, each bootstrapped replica will contain on average  $0.63N$  samples from the original dataset, some of them appearing multiple times. The remaining  $0.37N$  samples are often used for validation purposes.

Boosting goes one step further by training one learner after another and giving so far mis-classified examples a higher weight. This way learners trained in the beginning cover the general “easy” training examples and later trained learners become “experts” for certain cases. The final decision is made by a weighted majority voting, where the weight of each single learner is dependent on its performance on the complete dataset. Originally proposed as a meta-algorithm for classification, boosting can also be extended to regression problems (Friedman, 2001).

In the case of neural networks, even simply training the network multiple times on the same training set leads to some diversity because of the random initialization

of the network's weights and the random selection of patterns during learning. Other possibilities of introducing diversity into an ensemble of neural networks include varying the network's topology (number of hidden layers, number of neurons per layer, randomly sparse initialization of weight matrices (Zimmermann, Grothmann, Schäfer, and Tietz, 2006)), the learning algorithm, the learning rate, and regularization techniques like weight decay or early stopping.

### 5.3 Ensembles in Reinforcement Learning

Like supervised learning, RL can also benefit from ensembles. Surprisingly, the interest for ensemble methods in the RL literature has been quite low. There are a number of approaches that use ensembles to represent the value function (e.g., Singh, 1993; Tham, 1995; Sun and Peterson, 1999; Ernst, Geurts, and Wehenkel, 2005). The option of combining different policies has been explored less often (Jiang and Kamel, 2006; Wiering and van Hasselt, 2008), though with promising results. Faußer and Schwenker (2011) used ensembles for (online) TD learning with neural networks.

In the following, ways of combining different neural networks to an ensemble in an NFQ context are discussed.

#### **Combination of final policies**

A straightforward and, as we later shall see, quite successful way of creating a policy ensemble is letting each instance of the algorithm run for itself until a final policy or  $Q$ -function is determined and then combining those to obtain the final policy. This method was first (and independently of this work) proposed by Wiering and van Hasselt (2008). It makes no assumptions about the policy generating algorithm and is therefore suitable for combining algorithms that use a different notion of a  $Q$ -function (e.g., actor-critic algorithms) or no  $Q$ -function at all (e.g., methods that directly search the policy space (see Chapter 2)). A simple solution for combining policies is (weighted) majority voting. Wiering and van Hasselt (2008) propose and empirically evaluate a number of additional methods for combining policies. Those are rank voting, Boltzmann multiplication, and Boltzmann addition. In rank voting each ensemble member votes for each action, the voting weight depends on the rank the member assigns that action. Majority voting is a special case of rank voting where the voting function assigns all actions except the one considered best a voting weight of zero. Boltzmann multiplication and addition use Boltzmann distributions. Each ensemble member contributes such a distribution. For Boltzmann multiplication an action's probabilities are multiplied, while for Boltzmann addition they are added, to arrive at the probabilities of a final Boltzmann distribution that is then used for action selection. According to their experiments, Boltzmann multiplication and majority voting

work best.

### Selection of the “most agreeable” policy

Instead of combining several policies to one, one can as well select the presumably best policy of the ensemble. That policy could be the “most agreeable” one, i.e., the one that is most often among the majority (van Hasselt, 2010a).

### Ensemble representation of $Q$ -function

Instead of letting each instance run for itself, the ensemble can already be used to generate new targets in each iteration. To do so, after having trained all learners in an iteration, their combined output is used to generate new targets for the next iteration. E.g., the single outputs can be combined as a (weighted) average. This is similar to a number of previous approaches that use an ensemble as in a supervised learning setting (Singh, 1993; Tham, 1995; Sun and Peterson, 1999; Ernst, Geurts, and Wehenkel, 2005).

### Combination of final $Q$ -functions

When combining learners that use a  $Q$ -function, one can combine the single  $Q$ -functions to an ensemble  $Q$ -function and base the final policy on that. This can be achieved by, e.g., (weighted) averaging. Note that this is different from the ensemble representation of the value or  $Q$ -function.

### Combinations of Policies from a Single NFQ Run

As a computationally cheap alternative to multiple individual NFQ runs or multiple neural networks for representing the  $Q$ -function, one can form an ensemble using policies from successive NFQ iterations. Since one does not know when a good policy is found and thus when to stop iterating, assuming that after a minimum number of iterations there are more good than bad policies, an ensemble policy obtained this way is expected to be better than a randomly picked one.

Table 5.2 gives the complexity for each method relative to a standard NFQ run. The times needed for a single NFQ run as well as for calculating an action given a state using the resulting policy are considered constant. Each ensemble member needs this effort, therefore most methods need as much additional time as ensemble members  $k$  are added. Exceptions are the “most agreeable” policy approach, where a single policy is selected to be used during runtime, and the ensemble generated from a single NFQ run, where during policy generation only one NFQ run is needed.

In the following experimental section all methods mentioned above are evaluated, except the ensemble representation of the  $Q$ -function, because this method only

Method	Policy Generation	Runtime
Single NFQ run	$O(1)$	$O(1)$
Combination of final policies	$O(k)$	$O(k)$
Combination of final $Q$ -functions	$O(k)$	$O(k)$
“Most agreeable” policy	$O(k)$	$O(1)$
Ensemble representation of $Q$ -function	$O(k)$	$O(k)$
Policies from a single NFQ run	$O(1)$	$O(k)$

**Table 5.2:** Complexity of ensemble methods in NFQ.

addresses the problems related to supervised learning of the  $Q$ -function, the NFQ-specific problems (chattering, overestimation) remain.

## 5.4 Experiments

To evaluate the ensemble methods in an NFQ context, experiments were conducted using the pole-balancing, cart-pole, and wet-chicken benchmarks. In the following, the benchmark domains, the setup of the experiments, and the results are described.

### 5.4.1 Cart-Pole

The cart-pole benchmark is a control problem that has been studied in classical control theory as well as in the RL literature (first works in an RL context include Michie and Chambers, 1968; Barto, Sutton, and Anderson, 1983). The aim is to balance a pole attached to a cart by applying forces to the cart. The pole should remain in upright position and the cart not hit one of the boundaries of the track. The state space  $\mathcal{S} = (\theta, \dot{\theta}, x, \dot{x})$  is four-dimensional and contains the pole’s angle  $\theta$  and its angular velocity  $\dot{\theta}$  as well as the cart’s position  $x$  and its velocity  $\dot{x}$ . Possible actions are  $\mathcal{A} = \{-1, 0, 1\}$  and denote the force applied to the cart. Often, the actions are corrupted by a noise term. The dynamics are described by

$$f = af_m + \rho f_r \quad (5.11)$$

$$c = \frac{f + m_p l \dot{\theta}^2 \sin(\theta)}{m_t} \quad (5.12)$$

$$\ddot{\theta} = \frac{g \sin(\theta) - \cos(\theta)c}{\frac{1}{2}(\frac{4}{3} - \frac{m_F \cos(\theta)^2}{m_t})} \quad (5.13)$$

$$\ddot{x} = c - m_p l \ddot{\theta} \frac{\cos(\theta)}{m_t}, \quad (5.14)$$

	$g$	$m_c$	$m_p$	$l$	$f_m$	$f_r$	$\tau$
<b>Cart-Pole</b>	$9.8 \frac{\text{m}}{\text{s}^2}$	8 kg	2 kg	0.5 m	10 N	2 N	0.02 s
<b>Pole-Balancing</b>	$9.8 \frac{\text{m}}{\text{s}^2}$	8 kg	2 kg	0.5 m	50 N	10 N	0.1 s

**Table 5.3:** Parameters used for the cart-pole and pole-balancing benchmarks.

where  $f$  is the actually applied force,  $a$  the action,  $f_m$  a positive force constant,  $\rho$  a random variable distributed uniformly in  $[-1, 1]$ ,  $f_r$  a second positive force constant determining the amount of action noise,  $m_p$  the mass of the pole,  $l$  the length of the pole, and  $m_c$  the mass of the cart.

The state variables can be determined from the differential equations by numerical integration, for instance using Euler’s method (Hairer, Norsett, and Wanner, 2002):

$$x_{t+1} = x_t + \tau \dot{x}_t \quad (5.15)$$

$$\dot{x}_{t+1} = \dot{x}_t + \tau \ddot{x}_t \quad (5.16)$$

$$\theta_{t+1} = \theta_t + \tau \dot{\theta}_t \quad (5.17)$$

$$\dot{\theta}_{t+1} = \dot{\theta}_t + \tau \ddot{\theta}_t \quad (5.18)$$

$\tau$  denotes the duration of a timestep. Table 5.3 gives the parameters used in the experiments. Figure 5.3 illustrates the cart-pole problem.

In the RL setting of the cart-pole, a discrete reward signal is used as punishment when the pole falls or the cart moves outside the boundaries of the track:

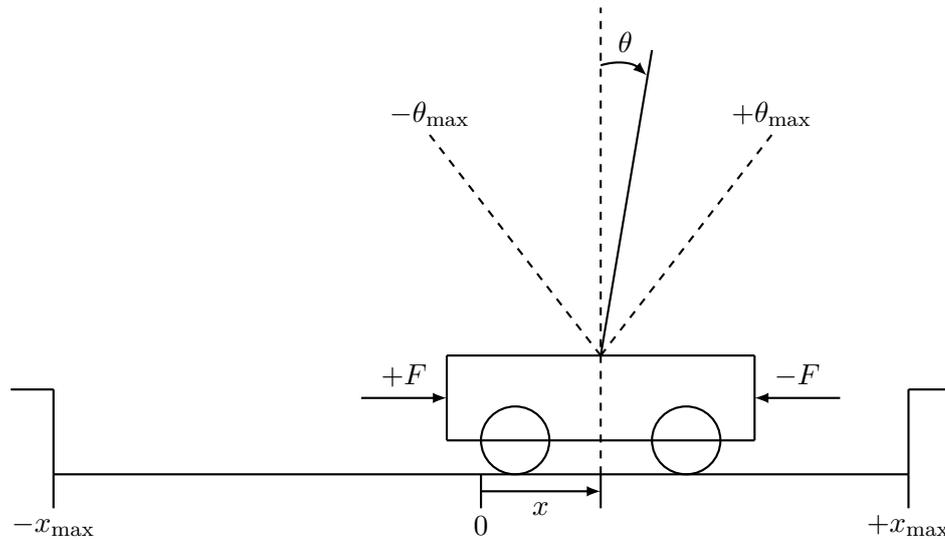
$$r_t^{01} := \begin{cases} 0 & \text{if } |x_t| < x_{\max} \wedge |\theta_t| < \theta_{\max}, \\ -1 & \text{otherwise.} \end{cases} \quad (5.19)$$

Additionally, a distance-based reward measure is defined with added bonus for the target region and punishment for states leading to failure:

$$r_t^d := \begin{cases} -|x_t| - |\theta_t| + 1 - 2x_t^2/0.05^2 + x_t^4/0.05^4 & \text{if } |x_t| < 0.05 \wedge |\theta_t| < 0.15, \\ -|x_t| - |\theta_t| - 10(|\theta_t| - 0.2) & \text{if } |\theta_t| > 0.2, \\ -|x_t| - |\theta_t| & \text{otherwise.} \end{cases} \quad (5.20)$$

## 5.4.2 Pole-Balancing

The pole-balancing benchmark is a variant of the cart-pole where only the angle and angular velocity are considered. Consequently, the only goal is to balance the pole. This problem is also known as the *inverted pendulum*. In one setting, an episode starts with the pole in upright position, the goal is then to balance the pole as long as possible. In another setting, an episode starts with the pole



**Figure 5.3:** Illustration of the cart-pole benchmark.

hanging downwards in its stable position; the goal is to apply forces in a way that cause the pole to swing up and then stabilize it.

For the experiments only the setting where an already upright pole has to be balanced were used. The parameters are given in Table 5.3.

### 5.4.3 Wet-Chicken

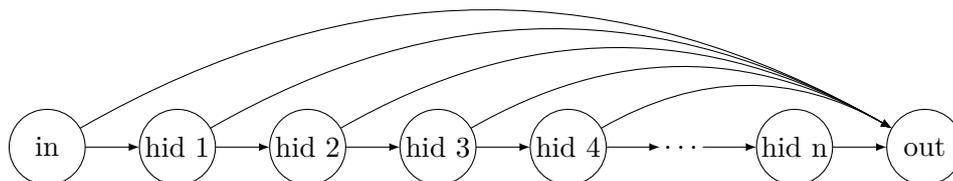
The wet-chicken benchmark was already used in Chapter 4. In contrast to the version used previously, here the canoe position  $x$  is not rounded to the nearest integer, but the continuous value is used directly, resulting in a continuous state problem.

### 5.4.4 Experimental Setup

For each domain, a number of observations were generated using random exploration.

For the pole-balancing benchmark, observations were generated in episodes. When applying actions randomly, the pole falls (and therefore the episode ends) after approximately six steps. Datasets were used of 25, 50, and 100 episodes, corresponding to 150, 300, and 600 observations. To assess a policy's quality, it was run 100 times for at most 3,000 steps. If a policy is able to balance the pole for 3,000 steps, it is considered successful.<sup>1</sup> For each dataset size, 50 distinct

<sup>1</sup>3,000 steps are chosen to get results comparable to those from Riedmiller (2005).



**Figure 5.4:** Deep, cascaded neural network where each layer is connected to the output layer. Each circle represents a layer of neurons, each arrow denotes a weight matrix realizing a full connection of the respective layers.

datasets were generated; an independent trial was run with each of them (i.e., 50 independent trials for each combination of method and dataset size).

For cart-pole, observation sets containing 10,000 and 30,000 observations were generated. Observations were generated starting from random positions with zero velocities and  $x \in [-2.3, 2.3]$ ,  $\theta \in [-0.2, 0.2]$ . For evaluation, the position was randomly initialized with  $x \in [-1, 1]$ . An episode was stopped when  $|x| > 2.4$  or  $|\theta| > 0.25$ . A policy’s quality is reported as average immediate reward from 100 episodes with a maximum length of 3,000 steps. For the cart-pole experiments, the distance-based reward function (5.20) was used. The results reported are averages of 40 completely independent trials, i.e., using 40 distinct datasets.

In the wet-chicken domain the datasets contained 500 observations and were generated with random exploration. Each run started at the beginning of the river and lasted for 500 steps. This was repeated 100 times to obtain 100 distinct datasets. The results reported are average rewards from 100 episodes with 1,000 steps each.

For all experiments the neural networks consisted of an input, two hidden, and an output layer (4-layer network). Each hidden layer contained 10 neurons. For the pole-balancing benchmark also a deep, cascaded architecture was used, where each layer is connected to the output as well (Figure 5.4). The deep, cascaded network contained 8 hidden layers (10-layer network), each hidden layer contained 10 neurons.

### 5.4.5 Results

The results of the experiments using the pole-balancing benchmark are shown in tables as number of successful policies. Table 5.4 gives the results using single networks, i.e., single policies obtained by a standard NFQ run as described above, as well as results from ensemble policies using majority voting. “4L” denotes the standard 4-layer network, “10LD” the deep, cascaded architecture. Table 5.5 gives the results of  $Q$ -averaging. The single network results are not included again, as the results of majority voting and  $Q$ -averaging are identical when only

	Number of Episodes		
	25	50	100
<b>1x 4L</b>	24/50 (48%)	20/50 (40%)	37/50 (74%)
<b>1x 10LD</b>	18/50 (36%)	22/50 (44%)	24/50 (48%)
<b>5x 4L</b>	28/50 (56%)	38/50 (76%)	43/50 (86%)
<b>10x 4L</b>	32/50 (64%)	37/50 (74%)	45/50 (90%)
<b>15x 4L</b>	33/50 (66%)	38/50 (76%)	46/50 (92%)
<b>20x 4L</b>	34/50 (68%)	37/50 (74%)	48/50 (96%)
<b>5x 10LD</b>	24/50 (48%)	26/50 (52%)	37/50 (74%)
<b>10x 10LD</b>	30/50 (60%)	35/50 (70%)	45/50 (90%)
<b>15x 10LD</b>	30/50 (60%)	37/50 (74%)	45/50 (90%)
<b>20x 10LD</b>	32/50 (64%)	40/50 (80%)	48/50 (96%)
<b>5x 4L &amp; 5x 10LD</b>	33/50 (66%)	36/50 (72%)	47/50 (94%)
<b>10x 4L &amp; 10x 10LD</b>	37/50 (74%)	43/50 (86%)	50/50 (100%)
<b>15x 4L &amp; 15x 10LD</b>	36/50 (72%)	43/50 (86%)	50/50 (100%)
<b>20x 4L &amp; 20x 10LD</b>	39/50 (78%)	45/50 (90%)	50/50 (100%)
<b>Riedmiller’s NFQ</b>	–	23/50 (46%)	44/50 (88%)

**Table 5.4:** Results for the pole-balancing benchmark from single policies (first two rows) and ensemble policies derived by **majority voting**. Given is the ratio of successful policies, i.e., policies able to balance at least 3,000 steps. The results from Riedmiller (2005) are given as a reference.

one network’s policy is used.

The performance of the present NFQ implementation when using 50 random episodes as training data approximately matches the performance reported by Riedmiller (2005). He does not give results for 25 episodes, here the performance for 100 episodes is somewhat worse. With more optimization of the learning process it would be possible to further improve the results for 50 and 100 episodes. In particular, for the network training procedure used here an adaption of the `num_epochs` parameter w.r.t. to the number of training examples seems to be crucial (a fixed value of 30 was used; experiments using `num_epochs = 15` for 100 episodes (not reported here) showed a notable improvement of single policy quality). However, when looking at the results of Table 5.4 it becomes obvious that by combining different networks to ensembles it is possible to match or even surpass the performance of an NFQ approach apparently fitting the pole-balancing problem better.<sup>2</sup>

<sup>2</sup>Apart from using VarioEta as opposed to Rprop, this thesis’s training process and the one used by Riedmiller (2005) have a further difference: While he used a fixed number of 100 epochs for training with Rprop (Riedmiller, 2010), here the MSE of the validation set is used to decide when to stop training.

	Number of Episodes		
	25	50	100
<b>5x 4L</b>	31/50 (62%)	36/50 (72%)	43/50 (86%)
<b>10x 4L</b>	31/50 (62%)	32/50 (64%)	48/50 (96%)
<b>15x 4L</b>	36/50 (72%)	31/50 (62%)	49/50 (98%)
<b>20x 4L</b>	34/50 (68%)	35/50 (70%)	50/50 (100%)
<b>5x 10LD</b>	24/50 (48%)	37/50 (74%)	36/50 (72%)
<b>10x 10LD</b>	31/50 (62%)	33/50 (66%)	40/50 (80%)
<b>15x 10LD</b>	29/50 (58%)	37/50 (74%)	41/50 (82%)
<b>20x 10LD</b>	30/50 (66%)	37/50 (78%)	42/50 (84%)
<b>5x 4L &amp; 5x 10LD</b>	30/50 (60%)	31/50 (62%)	43/50 (86%)
<b>10x 4L &amp; 10x 10LD</b>	31/50 (66%)	39/50 (78%)	43/50 (86%)
<b>15x 4L &amp; 15x 10LD</b>	31/50 (66%)	40/50 (80%)	44/50 (88%)
<b>20x 4L &amp; 20x 10LD</b>	33/50 (66%)	40/50 (80%)	44/50 (88%)

**Table 5.5:** Results for the pole-balancing benchmark from ensemble policies derived by **Q-averaging**.

	Number of Episodes		
	25	50	100
<b>5x 4L</b>	25/50 (50%)	24/50 (48%)	43/50 (86%)
<b>10x 4L</b>	25/50 (50%)	32/50 (64%)	41/50 (82%)
<b>15x 4L</b>	25/50 (50%)	29/50 (58%)	43/50 (86%)
<b>20x 4L</b>	27/50 (54%)	32/50 (64%)	45/50 (90%)

**Table 5.6:** Results for the pole-balancing benchmark of “**most agreeable**” policies.

Adding networks to the ensemble increases the performance to a certain point, which is not always reached here (adding even more networks than 20 would be required). Among networks of the same type there seems to be already enough diversity to benefit from an ensemble, but combining networks of different types is better—not only are the heterogeneous ensembles containing the most members (15x 4L & 15x 10LD and 20x 4L & 20x 10LD) better than all homogeneous ensembles, in 11/12 cases heterogeneous ensembles perform better than homogeneous ones of the same size.

Comparing the aggregation techniques, majority voting is superior to  $Q$ -averaging. While for the ensembles of 4L networks both perform equivalently, for combination of 10LD networks and the heterogeneous ensembles majority voting is clearly better (8/12 and 12/12 cases, respectively). A reason for this might be that the different networks’  $Q$ -functions have different ranges. Another reason for majority voting being superior might lie in the fact that a single really bad  $Q$ -function can dominate the average (drastically decreasing or increasing it); with majority

	Number of Episodes		
	25	50	100
1x 4L/5	25/50 (50%)	28/50 (56%)	44/50 (88%)
1x 4L/10	30/50 (60%)	28/50 (56%)	45/50 (90%)
1x 10LD/5	24/50 (48%)	24/50 (48%)	35/50 (70%)
1x 10LD/10	22/50 (44%)	31/50 (62%)	36/50 (72%)
2x 4L/5	26/50 (52%)	33/50 (66%)	44/50 (88%)
4x 4L/10	33/50 (66%)	39/50 (78%)	48/50 (96%)
10x 4L/10	34/50 (68%)	39/50 (78%)	49/50 (98%)
2x 10LD/5	28/50 (56%)	26/50 (52%)	38/50 (76%)
4x 10LD/10	33/50 (66%)	41/50 (82%)	47/50 (94%)
10x 10LD/10	33/50 (66%)	43/50 (86%)	48/50 (96%)
1x 4L/5 & 1x 10LD/5	34/50 (68%)	33/50 (66%)	42/50 (84%)
2x 4L/10 & 2x 10LD/10	36/50 (72%)	40/50 (80%)	48/50 (96%)
5x 4L/10 & 5x 10LD/10	41/50 (82%)	45/50 (90%)	50/50 (100%)

**Table 5.7:** Results of **majority voting** with policies from **successive iterations** for the pole-balancing benchmark. For these experiments policies from successive iterations of a single NFQ run were used. The number of policies from successive iterations is noted after the slash. For example, “2x 4L/5” denotes an ensemble containing policies from the last five iterations of two independent NFQ run using a 4-layer network (total of 10 ensemble members).

voting, the bad  $Q$ -function has only one vote, the magnitude of the  $Q$ -values plays no role.

Table 5.6 contains results of the “most agreeable” policy for the pole balancing benchmark. While for all observation sizes the single selected policy cannot reach the performance of an ensemble of the policies selected from, the policy selected as the “most agreeable” one performs better than a single randomly selected policy. Therefore, this method might be worthwhile considering when only the increased complexity for policy generation can be accepted, but the increased complexity of a full ensemble during runtime (policy execution) cannot.

Table 5.7 shows results of experiments with policies from successive iterations from single NFQ runs and combinations of successive iterations from independent NFQ runs. Combinations of policies from NFQ runs using the 4-layer network as well as the 10-layer deep network were also evaluated. In general, using policies from successive iterations in ensembles improves the performance and the results are comparable to those from ensembles containing policies from independent NFQ runs (Table 5.4). However, here the ensembles need to be much larger to achieve similar results. For example, to achieve 45/50 successful policies with 50 episodes of observations, an ensemble containing 100 members is needed here (5x 4L/10 & 5x 10LD/10), while an ensemble of 40 policies from individual runs

Observations		Cart-Pole		Wet-Chicken
		10,000	30,000	500
Single	1x 4L	$-0.35 \pm 0.07$	$-0.140 \pm 0.040$	$12.9 \pm 0.2$
Successive iterations	1x 4L/5	$-0.32 \pm 0.08$	$-0.089 \pm 0.029$	$13.4 \pm 0.2$
	1x 4L/10	$-0.21 \pm 0.04$	$-0.146 \pm 0.043$	$13.3 \pm 0.2$
Independent runs	5x 4L/1	$-0.11 \pm 0.02$	$-0.030 \pm 0.005$	$13.3 \pm 0.2$
	10x 4L/1	$-0.07 \pm 0.02$	$-0.021 \pm 0.002$	$13.3 \pm 0.2$
Combined	5x 4L/2	$-0.09 \pm 0.02$	$-0.026 \pm 0.003$	$13.4 \pm 0.2$
	5x 4L/5	$-0.07 \pm 0.01$	$-0.022 \pm 0.002$	$13.5 \pm 0.2$
	5x 4L/10	$-0.05 \pm 0.01$	$-0.019 \pm 0.002$	$13.4 \pm 0.2$
	10x 4L/2	$-0.07 \pm 0.01$	$-0.020 \pm 0.002$	$13.4 \pm 0.2$
	10x 4L/5	$-0.06 \pm 0.01$	$-0.019 \pm 0.001$	$13.4 \pm 0.2$
	10x 4L/10	$-0.05 \pm 0.01$	$-0.017 \pm 0.001$	$13.4 \pm 0.1$

**Table 5.8:** Experimental results using policies from a single NFQ run and ensemble policies from a number of final successive iterations from single NFQ runs, completely independent runs, and combinations of both. The numbers shown correspond to the average immediate reward (from 40 (cart-pole) and 100 trials (wet-chicken), respectively). For cart-pole average rewards instead of ratios of successful policies (as for pole-balancing) are given, since most policies are able to complete the required steps; the distance-based reward is therefore a more meaningful measure, as it also captures how well a policy keeps the cart in the middle of the track and the pole upright.

suffices as well (20x 4L & 20x 10LD).

Table 5.8 contains the results of the experiments using the cart-pole and wet-chicken benchmarks. Here, only experiments using 4-layer networks were performed. Again, in addition to ensembles consisting of policies from independent NFQ runs (denoted “independent runs”), also policies from successive iterations of a single NFQ run (“successive iterations”) as well as combinations of both approaches (“combined”) are included. In the cart-pole domain, ensembles of successive iterations do not significantly increase the performance. The results for wet-chicken suggest a slight improvement of ensembles from successive iterations. Ensembles from independent NFQ runs lead to significantly better policies in the cart-pole domain and tend to improve the wet-chicken policies as well. In the experiments performed here, the best results are obtained by the combined approach, using ensembles consisting of policies from a number of successive runs from multiple independent runs.

Overall, it seems advisable to focus on independent NFQ runs. Ensembles containing policies from individual runs in general deliver better results, as they are less correlated and can therefore contribute more to an ensemble. Nonetheless, ensembles of policies from successive iterations almost always lead to better performance, albeit not as pronounced as the performance increase when using

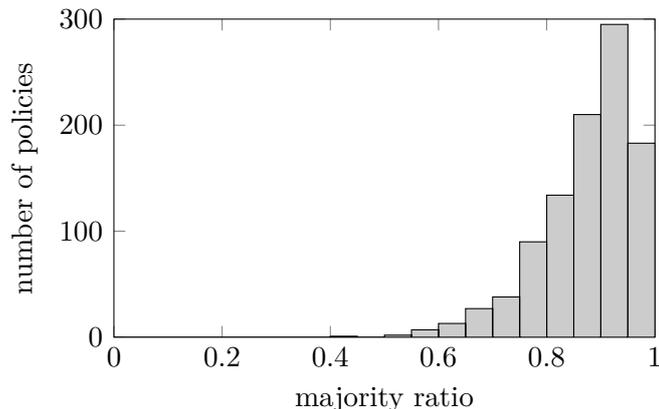
policies from successive runs. The preferred solution for a particular application therefore depends on the concrete requirements: If one cannot afford many off-line NFQ runs, but evaluating a large ensemble at runtime is feasible, one should use policies from successive iterations. If, on the other hand, the computational resources at runtime are limited and the user can only afford a rather small ensemble, one should use policies from independent NFQ runs to build the ensemble, as the performance-per-ensemble-size ratio of ensembles from individual runs is better. Of course, a combination of the two approaches is possible as well.

## 5.5 Why Do Ensembles of NFQ Policies Work?

To understand why ensembles of NFQ policies improve upon the performance of single policies, consider the three problems named by Dietterich (2000) again: the statistical problem (a solution that fits both, training and validation data, can still deviate from the true underlying function), the computational problem (since an exhaustive search of the hypothesis space is typically intractable, methods like stochastic gradient descent are used that can only approximate the best hypothesis), and the representational problem (the function approximator might be unable to represent the actual function, because it lies outside the hypothesis space). All three problems potentially contribute to a learner’s failure in classic classification and regression problems. An NFQ run comprises a series of regression problems, leading to an even more intricate situation.

If the members of the ensemble are accurate and diverse, i.e., make (in expectation) better decisions than random guessing and do not make the same errors, combining them can “average out” the individual errors and lead to a better ensemble solution. If one deals with a binary classification problem, a classifier  $h_i$  is better than random guessing if its error rate  $e_i < 0.5$ . If we have three of those classifiers, for a given sample two or more have to be wrong for the majority decision to be wrong. With an increasing number of members, the probability of the majority being wrong decreases. If the ensemble members’ errors were completely uncorrelated, the ensemble performance could be increased indefinitely by adding more and more members. However, in practice some correlation will always be present, resulting in an upper limit for the reasonable ensemble size.

Ensembles of policies in RL work for similar reasons as described above. In each state the decision must be made what action to choose. If the individual policies are accurate and diverse, they will choose a good action most of the time and will choose differently for some states. Using the policies in an ensemble then again allows “averaging out” the errors of individual policies (at least to some extent). Additionally, and maybe even more importantly, ensembles allow to suppress very poor policies. Figure 5.5 illustrates how often a policy is among the majority (data from pole-balancing ensembles containing 20 policies (4L networks) trained with observations from 100 episodes). If a policy is always among the majority, i.e.,



**Figure 5.5:** Histogram of majority ratios. Ensembles with 20 members from the pole-balancing experiments with 100 episodes were used and for each policy it was determined how often it is among the majority of its ensemble. This was done for 50 trials, so the histogram contains values from  $50 \cdot 20 = 1000$  policies.

has a majority ratio of 1, it behaves exactly like the ensemble and could be used instead. The histogram shows that most policies are often among the majority. On the other hand, there is a considerable amount of policies with a majority ratio of 0.7 and less. Although those policies choose the majority action most of the time, 30% or more of their choices are overruled by other ensemble members. The few policies with a majority ratio of less than 0.5 represent those instances where NFQ completely failed; they occasionally choose the optimal action merely by chance. When executed alone their performance would be very poor, but in the ensemble they are overruled most of the time.

That the aspect of “averaging out” individual errors contributes as well can be seen from the fact that not a single policy is always among the majority. Only 11 out of the  $20 \cdot 50 = 1000$  policies used for the histogram in Figure 5.5 have a majority ratio of 0.99 or greater. The performance of those single policies will hardly be distinguishable from the ensemble performance, but they are quite rare—on average the ensemble of only every second trial contains such a policy. Moreover, it is hard to identify those policies. The experimental results also showed that on average “most agreeable” policies, i.e., those most often among the majority, do not perform as well as the corresponding full ensembles. This supports the assumption that the ensemble members “average out” each other’s errors.

## 5.6 Continuous Actions

So far the discussion was limited to discrete-action problems. This section will discuss ideas for ensembles for continuous-action problems, which can serve as

starting points for further research.

Majority voting appears to be the best approach for discrete actions. Unfortunately, implementing majority voting for continuous actions is not straightforward. Consider an ensemble of continuous-action policies. For a given state, member  $i$  outputs a vector  $\vec{a}_i \in \mathbb{R}^{D_{\mathcal{A}}}$ , with  $D_{\mathcal{A}}$  the dimensionality of the action space. Every  $\vec{a}_i$  is a point in the action space. In analogy to the discrete action case, we expect the majority to output an action that in some sense is “near” the optimal action. The outputs of the members that are wrong will be scattered. Now the question arises how to first identify this majority and second determine a final ensemble action vector from the members’ actions.

Assuming a metric space, e.g., Euclidean, we can determine the pair-wise distances between the members’ actions. Given further a distance parameter  $d_{\max}$ , we can for each member’s action determine the set of neighbors within a maximum distance of  $d_{\max}$ , i.e., for member  $i$  the set is defined as

$$N_i^{d_{\max}} := \{\vec{a}_j \mid \text{dist}(\vec{a}_i, \vec{a}_j) < d_{\max} \wedge i \neq j\}, \quad (5.21)$$

with  $\text{dist}(\cdot, \cdot)$  the distance function for the assumed metric space. From the sets one determines the one with most members and chooses its action as ensemble action, i.e.,

$$\pi^{d_{\max}} := \vec{a}_i, i = \arg \max_j |N_j^{d_{\max}}|. \quad (5.22)$$

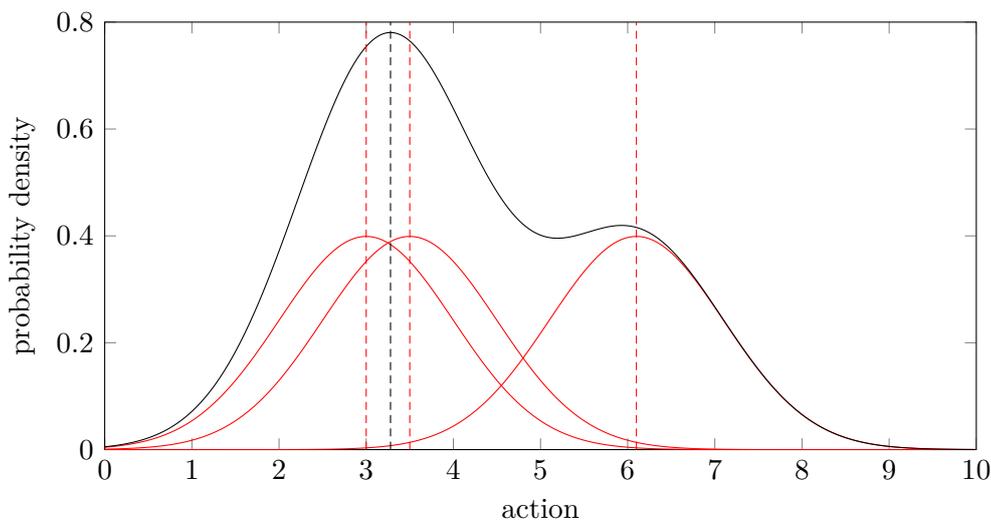
If there is not just one action with the maximum number of neighbors, the final action can be determined by randomly selecting from the set of actions with the maximum number of neighbors (maximum set). It is also thinkable to choose the action that lies in the maximum set’s centroid.

Another idea is to add Gaussian functions, one for each ensemble member with the center at the member’s action’s point. If we superimpose more than two Gaussians, the resulting function will in general have exactly one maximum, which we then consider the chosen action.<sup>3</sup> Finding the maximum analytically is non-trivial, though. As a solution, it is proposed to use gradient ascent, starting from the centers of the Gaussians, i.e., from the actions proposed by the members. While this approach will only find local maxima for some starting points, there will be at least one starting point from which the global maximum can be found. See Figure 5.6 for an example.

For the Gaussian function approach the parameter corresponding to the standard deviation in a Gaussian density function would serve as the distance parameter.

---

<sup>3</sup>There are cases with the actions positioned in a symmetrical way that lead to a maximum for each action. For example, two-dimensional actions being positioned equidistantly on a circle would yield such a situation. However, in practice this will happen very rarely; it could be dealt with by random action selection.



**Figure 5.6:** Example of superimposition of Gaussian functions, here for one-dimensional continuous actions and three ensemble members. For each action proposal by an ensemble member, a Gaussian function is added (red), resulting in the black curve. The proposed actions are  $a_0 = 3$ ,  $a_1 = 3.5$ , and  $a_2 = 6.1$ . The maximum of this curve can be determined by gradient ascent from the proposed actions. While doing gradient ascent from the very right action  $a_2$  would lead to a local maximum, starting from any of the other two actions will lead to the global optimum at  $a = 3.279$ . The distance parameter is chosen  $\sigma = 1$ .

## 5.7 Summary and Conclusion

In this chapter, the combination of ensemble methods with NFQ was explored. After an introduction to NFQ, problems contributing to its reliability issues were named. By using ensembles, these issues can be addressed. A brief overview of ensemble methods was given, mentioning the importance of accuracy and diversity (or ambiguity) of the ensemble members. It followed an evaluation of the methods on the pole-balancing, cart-pole, and wet-chicken benchmarks. In general, large and diverse ensembles perform best, but also small ensembles lead to improvement. The recommended ensemble size depends on what the user can afford computationally. The more ensemble members already present, the less improvement can be expected from adding a member. For the problems considered here, ensemble sizes of about 20 members seem reasonable. Regarding the ensemble methods, majority voting can be recommended, since it lead to excellent results and its implementation is straightforward. Finally, possibilities for ensembles for continuous actions were discussed, which can serve as a starting point for further research.

Overall, ensembles improved the quality of the resulting policies in each of the considered domains. In general, the author expects ensembles to be always at

least as good as randomly selected policies; if the quality of the policies to choose from varies, ensembles will likely perform better for reasons similar to the ones leading to the superiority of ensembles in classification problems. When using ensembles in an NFQ context, the suppression of poor policies seems to be an important property. However, “averaging out” of individual errors contributes as well (as confirmed by the better results of full ensembles over “most agreeable” policies).

If the user can afford the computational effort, large and diverse ensembles, possibly containing policies from successive iterations, are the most promising approach. Furthermore, once a regular NFQ implementation is available, the extension to the ensemble methods presented in this chapter is almost trivial, in particular using majority voting with policies from independent NFQ runs. The same holds for parallelizing the methods, both for policy generation and execution, taking advantage of the recent developments of commodity hardware, shifting from single to multi-core architectures (Borkar et al., 2005).

# 6

## Self-Assessment in Continuous Domains

This chapter investigates methods for self-assessment in continuous domains. In Chapter 4, Section 4.2, we discussed how different policies can be compared in a discrete setting by considering not only the value function, but also its uncertainty. We have seen that considering the value function alone can be misleading. If one considers the value function’s uncertainty as well, a more realistic assessment of policy quality becomes possible. Ideally, in a continuous setting one would do the same: “somehow” determine a value function along with its uncertainty and then compare policies using their values minus the  $\xi$ -weighted uncertainty. Unfortunately, this is not possible, since there is no way of determining the uncertainty in a way similar to what was used in the discrete case. Straightforward policy evaluation, however, works much better for continuous problems with function approximation than the table-based discrete variant. We will see that using alternative methods or different data for policy evaluation the correlation between estimated and true policy performance and thus the quality of the policy performance estimation can be improved drastically.

Throughout the chapter policies from the experiments described in the previous chapter will be used (for pole-balancing a different reward function was used; more details in Section 6.3).

### 6.1 Value Function-Based Self-Assessment

If the value function is available, one can use the expected return as a measure for the quality of a policy  $\pi$ , here for the continuous state space  $\mathcal{S}$ ,

$$J^\mu(\pi) = \int_{\mathcal{S}} \mu_0(s) V^\pi(s) ds, \quad (6.1)$$

where  $\mu_0(s)$  denotes the probability density of  $s$  being a start state. This integral can be approximated by averaging the values of the start states  $S_0$  found in the observation set:

$$J^{\mu, S_0}(\pi) = \frac{1}{|S_0|} \sum_{s \in S_0} V^\pi(s) \quad (6.2)$$

Unfortunately, the exact value function of a policy  $V^\pi$  is usually unknown and must somehow be estimated from observations. In fact, when dealing with policies generated by a value function based method like neural fitted  $Q$ -iteration (NFQ), an estimation of the value function is already available in form of the  $Q$ -function that was generated when determining the policy. However, it will show that, although there is some correlation between the  $Q$ -function and the actual policy performance, one can do much better by determining a new  $Q$ -function using policy evaluation.

## 6.2 Fitted Policy Evaluation

To determine the  $Q$ -function of a given policy and a set of observations of a Markov decision process (MDP), one can use an algorithm similar to fitted  $Q$ -iteration (FQI), called fitted policy evaluation (FPE). FPE can be considered as the sample-based equivalent to policy evaluation using dynamic programming (DP), as described in Chapter 2, Section 2.4.1. Recall the DP update step for policy evaluation:

$$Q_{k+1}^\pi(s, a) := \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a) [\mathcal{R}(s, a, s') + \gamma Q_k^\pi(s', \pi(s'))] \quad (6.3)$$

As with FQI, when using a sample-based approach, the expectation realized by the sum over successor states and the corresponding transition probabilities  $\mathcal{P}$  are approximated by the samples. Thus, the update of targets in an iteration of FPE is done using

$$Q_{k+1}^\pi(s_i, a_i) := r_i + \gamma Q_k^\pi(s'_i, \pi(s'_i)) \quad \forall i \in \{1, \dots, M\}, \quad (6.4)$$

where  $(s_i, a_i, s'_i, r_i)$  is an observation tuple and  $M$  the number of observations.

Algorithm 8 shows the complete FPE procedure. The only difference to FQI is in line 7—instead of maximizing the  $Q$ -value of a successor state over the actions, the  $Q$ -value of the action selected by policy  $\pi$  is used. As with FQI, this can be combined with any function approximator.

Using FPE one can obtain a value function for a given policy and then use this value function to obtain the expected return  $J^{\mu, S_0}$  (using (6.2)) of the policy as a quality measure. However, as stated above, if the policy was generated using FQI, one already has an estimate of the policy's value function. Unfortunately, as we will see in the next section, it is not a very good one.

So far the procedure is very similar to the discrete case; the only difference lies in the way of determining the value function—in the discrete setting, DP-based policy evaluation was used, here we use (sample-based) FPE. The next step would consist in determining the uncertainty. Analogous to the discrete case, a Monte Carlo approach comes to mind, here using ensembles. Instead of performing

**Algorithm 8:** Fitted Policy Evaluation**Input:** set of observations  $\mathcal{O} = \{(s_i, a_i, r_i, s'_i) | i = 1, \dots, M\}$ ,  $\pi$ ,  $\gamma$ **Result:**  $Q$ -function estimate of  $Q^\pi$ 


---

```

1 begin
   | inputs are state-action pairs from observations
2   inputi := (si, ai)  ∀i ∈ {1, ..., M}
   | set rewards as initial targets
3   targeti := ri  ∀i ∈ {1, ..., M}
4   k := 0
5   while stopping criteria not reached do
   |   train function approximator to map input ↦ target
6   |   Qkπ := train(input, target)
   |   determine new targets
7   |   targeti := ri + γQkπ(s'i, π(s'i))  ∀i ∈ {1, ..., M}
8   |   k := k + 1
9   return Qkπ

```

---

FPE only once, it is performed multiple times, each time delivering a new value function sample. Unfortunately, on second thought it becomes obvious that this method would not estimate the uncertainty contained within the data. Instead, it would give the uncertainty of the complete learning process from input data (observation set) to output data (value function). For example, consider neural FPE with the training data being bootstrapped replica of the observation set, i.e., if the observation set contains  $N$  samples, we repeatedly sample (with equal probability) from the original set until we have  $N$  samples. Even using the same observation set, a Monte Carlo approach based on differently sized networks (e.g., different number of hidden neurons) will lead to different uncertainty estimates. A large network will lead to a higher uncertainty estimate than a smaller one, because its flexibility and thus its variance are higher (and its bias lower). The major difference between the discrete and continuous settings is the following: In the discrete setting, one uses an explicit model of the MDP. The Dirichlet distributions constitute a distribution of MDPs, which can be used to sample MDPs. In the continuous case, the observation set corresponds to a single MDP, a distribution over MDPs we do not possess. By using the observations to build a probabilistic model, for example using Gaussian processes (Rasmussen and Williams, 2006), and then sampling from that model, a similar approach would become possible. However, this would require making additional, possibly problem-specific assumptions, whereas the MDP model consisting of transition probabilities and reward expectation is a natural choice for any discrete MDP. The idea of direct uncertainty estimation for continuous domains is therefore abandoned; instead, other methods for estimating a policy's performance are evaluated.

### 6.3 Correlation Between True Performance and Value Function Estimate

To assess the quality of a value function as an indicator of the true performance of a policy, one can use the correlation coefficient, which is a measure of the linear dependency of two random variables. In the present case, those two random variables are the true performance of a policy  $\pi$  and an estimate  $\hat{J}^{\mu, S_0}(\pi)$  of its expected return. Ideally,  $\hat{J}^{\mu, S_0}(\pi)$  gives the true expected return, i.e., the expected sum of discounted rewards when running  $\pi$  on the MDP. If for two policies  $\pi$  and  $\pi'$   $\hat{J}^{\mu, S_0}(\pi) > \hat{J}^{\mu, S_0}(\pi')$ ,  $\pi$  can be considered the better policy, since it is able to collect a greater sum of rewards. If  $\hat{J}^{\mu, S_0}(\pi)$  fails at giving the true return of  $\pi$ , which for real applications it often does due to effects such as chattering, overestimation of  $Q$ -values, and under- and over-fitting, one can at least hope that  $\hat{J}^{\mu, S_0}(\pi)$  is suitable as a basis for deciding whether one policy is better than another. For that to work, a common basis is necessary, i.e.,  $\hat{J}^{\mu, S_0}(\pi)$  should be determined using the same method and the same data for each policy.

FPE is may be used in policy iteration, where one tries to find a near-optimal policy by repeatedly doing policy evaluation and policy improvement. To actually evaluate a final policy without a subsequent policy improvement step has been independently proposed by Migliavacca et al. (2010). They used tree-based FPE to evaluate candidate policies in a direct policy search setting (see Migliavacca et al. (2011) for an extended version).

For the experiments in Chapter 5 ensembles of policies were generated using NFQ with a common observation set for each ensemble. For the experimental results in this chapter, for wet-chicken, the observation sets and corresponding policies from Chapter 5 were employed. For pole-balancing, a modified reward function was used. Consequently, new observation sets and policies had to be generated. The modified reward function not only punishes failure, but also includes a bonus of 1 when  $|\theta| < 0.03$ , i.e., the pole is almost upright. This way also policies able to balance the pole all the time are distinguishable (with the original reward function they would lead to an average reward of 0, no matter how well they balance the pole).

Using the final  $Q$ -function of each NFQ run,  $\hat{J}^{\mu, S_0}(\pi)$  of each policy  $\pi$  of the ensemble was determined. From this the correlation coefficient between the true mean reward  $\bar{r}_i, i = 1, \dots, K$  and  $\hat{J}^{\mu, S_0}(\pi_i), i = 1, \dots, K$ , with  $K$  the number of ensemble members, was calculated (for each ensemble separately). Thus a correlation coefficient for each ensemble was obtained.  $\hat{J}_{\text{NFQ}}^{\mu, S_0}$  was determined based on the  $Q$ -function from the NFQ runs and  $\hat{J}_{\text{NFPE}}^{\mu, S_0}$  based on value functions from a separate neural FPE run for each ensemble member. Table 6.1 shows the correlation coefficients of the actual policy performances (mean reward) and the estimates  $\hat{J}^{\mu, S_0}(\pi)$  using the  $Q$ -functions from the NFQ run (*Original NFQ*) and neural FPE (*NFPE*). Except for the pole-balancing experiments with observations from

Benchmark	Correlation	
	Original NFQ	NFPE
Pole-Balancing, 50 episodes	$-0.02 \pm 0.02$	$0.31 \pm 0.02$
Pole-Balancing, 100 episodes	$0.14 \pm 0.02$	$0.33 \pm 0.02$
Wet-Chicken	$-0.03 \pm 0.03$	$0.17 \pm 0.05$

**Table 6.1:** Correlations between true policy performance and value functions. *Original NFQ* denotes the performance’s correlation with  $\hat{J}_{\text{NFQ}}^{\mu, S_0}$ ; *NFPE* denotes the correlation with  $\hat{J}_{\text{NFPE}}^{\mu, S_0}$ .

100 episodes the value function from the original NFQ run is completely uninformative with a correlation of approximately zero. When more observations are available (100 episodes), the situation becomes better. However, when doing FPE, where the policy is fixed throughout the run, the results are much better with correlation coefficients around 0.3 for the pole-balancing benchmark and 0.2 for wet-chicken.

Reasons for the superiority of the FPE approach include the absence of chattering as well as the maximum bias when the policy is fixed. Furthermore, the task of learning the  $Q$ -function for a given fixed policy is easier than learning the policy and the value function at the same time, which is done in value iteration variants like NFQ.

## 6.4 Different Function Approximator and Different Data

Further improvements can be achieved by using a different function approximator or a different dataset to do FPE. If the dataset is sufficiently large to avoid under- and over-fitting, FPE should be able to determine an approximation of the value function that is close to the true one, given that the function approximator is powerful enough. In practical applications, however, the amount of available observations is often limited. So far for FPE the same dataset was used that previously served as the basis for policy generation. Similar to flawed self-assessment of humans, where the least competent often are also the most unaware of their own incompetence (Kruger and Dunning, 1999), using the same data and the same function approximator to assess a policy can be disadvantageous—if the data mislead the function approximator to produce an inferior policy, chances are it is mislead again when doing policy evaluation. In the following, two ways to circumvent this problem will be discussed and evaluated.

The most natural solution is to use a different dataset for FPE than was used to determine the policy to evaluate. Of course, this requires that additional data is available and implies a trade-off—what portion of the data should be used to determine the policy and what to evaluate it? For simplicity, we evaluate only

Benchmark	Correlation		
	NFPE <sub>diff</sub>	TreeFPE	TreeFPE <sub>diff</sub>
Pole-Balancing, 50 episodes	0.38 ± 0.02	0.29 ± 0.02	0.27 ± 0.02
Pole-Balancing, 100 episodes	0.40 ± 0.02	0.46 ± 0.02	0.48 ± 0.02
Wet-Chicken	0.26 ± 0.02	0.63 ± 0.05	0.55 ± 0.05

**Table 6.2:** Correlation coefficients between true policy performance and  $\hat{J}^{\mu, S_0}$  for the pole-balancing and wet-chicken benchmarks. For *NFPE<sub>diff</sub>* neural FPE on a different dataset was used. *TreeFPE* and *TreeFPE<sub>diff</sub>* denote extra-tree-based FPE on the same and a different dataset, respectively.

the case where the size of both datasets is equal.

Another possible solution lies in the use of an alternative function approximator for FPE. It should be powerful enough to be able to represent the value function. At the same time it is advisable to choose a function approximator that is as different as possible from the one used to generate the policy. The idea is that the more different the function approximators are, the less likely they are to make the same mistakes. This is related to the diversity issue of ensembles, where the different ensemble members should all be accurate (powerful) and diverse (different) (Krogh and Vedelsby, 1995; Dietterich, 2000). It also resembles ideas from safety-critical applications, where for a certain task not just one, but multiple different implementations are used (Lala and Harper, 1994). The more diverse the implementations are, the less likely is a failure of all of them for a given situation.

Here, extremely randomized trees (extra-trees) (Geurts, Ernst, and Wehenkel, 2006) take the role of the alternative function approximator. As a method based on regression trees, extra-trees are quite different from neural networks. At the same time, they are known to produce good solutions if a reasonable amount of training data is available. As an additional advantage, generating extra-trees is computationally quite cheap compared to backpropagation training of a neural network.

So in addition to neural FPE using the same data that was used for policy generation, neural FPE was performed using a different set of observations of the same size. Tree-based FPE as well was run using the same and a different dataset. The correlation coefficients of the resulting  $\hat{J}^{\mu, S_0}$  functions are summarized in Table 6.2.

In comparison with the NFPE results based on the same data, the correlation can always be improved, especially for the wet-chicken benchmark. Extra-tree based FPE (TreeFPE) achieves an even better correlation, except for the pole-balancing benchmark using observations from 50 episodes. When using as little as 50 episodes, the limited predictive power of trees hinders them in finding a better fitting value function than the neural network based approach. When more data

are available, however, they achieve a better correlation because of their different nature, no matter if they are trained on the same data used for policy generation or a different dataset.

In summary, one can conclude that NFQ leads to  $Q$ -functions that carry only little information about the quality of their respective policies: When following the  $Q$ -function's policy, i.e., always choosing the action that maximizes the  $Q$ -function, the obtained return will most likely be different from the one the  $Q$ -function indicates. To obtain a  $Q$ -function that is closer to the true return and is therefore more appropriate to assess a policy's quality, one can use fitted policy evaluation, ideally using a separate dataset or a different function approximator.

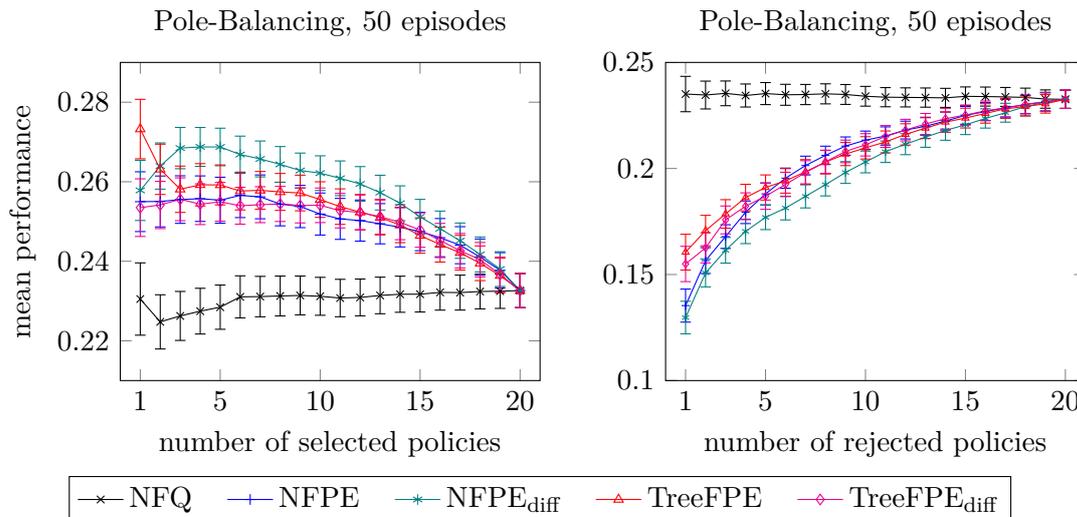
## 6.5 Policy Selection and Rejection

One application of an estimate of the expected return of a policy is policy selection or rejection. Given a set of policies, the aim is to order them by their performance. The resulting ordered list can then be used to select a number of good or reject a number of poor policies.

For the experiments in this section, the policies from the pole-balancing and wet-chicken benchmarks were ordered using different estimates of  $J^{\pi, S_0}$ . Then a number  $n$  of policies from the list was selected, with  $n = 1, 2, \dots, K$ , where  $K$  is the total number of policies to be compared. For the selected set the true mean performance was determined. When rejecting policies, the list is ordered the other way around, starting with the smallest estimate of  $J^{\pi, S_0}$ . Again, an increasing number  $n$  of policies is selected, starting from the top of the list, and the mean performance of the selected set is determined. In the case of rejection, the mean performance of policies that would be rejected is determined.

In essence, this is the same as what was done in Chapter 4, Section 4.2 for discrete domains.

Figures 6.1 and 6.2 show the results for the pole-balancing benchmark using policies generated from 50 and 100 random episodes, respectively. For the policies from 50 episodes, the estimate of the expected return based on the NFQ value function is completely uninformative, one could as well select a policy at random. Given the correlation coefficient of approximately zero, this comes as no surprise. All other approaches work comparably in this setting. When using observations from 100 episodes, the original NFQ  $Q$ -function also has some correlation with the true performance and makes it possible to select policies that are better than the average or reject policies worse than the average. Other approaches, however, perform considerably better. Although the linear correlation between  $\hat{J}_{\text{NFPE}}^{\mu, S_0}$  and the true performance is better than the correlation with  $\hat{J}_{\text{TreeFPE}}^{\mu, S_0}$ , when actually selecting good or poor subsets of policies, the tree approach performs consistently better. Using different data, a further slight improvement is possible. In the pole-balancing experiments also the advantage of using a different dataset in

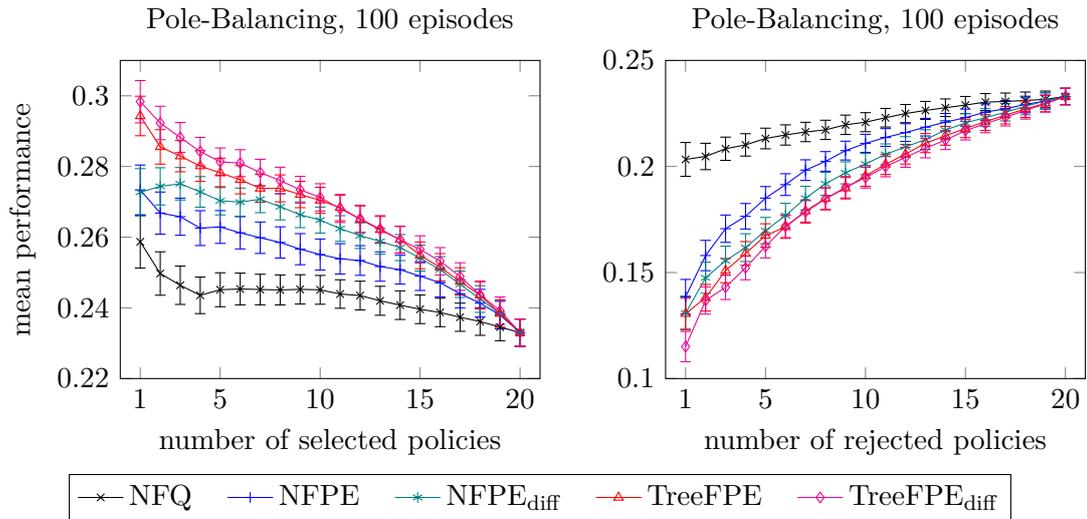


**Figure 6.1:** Mean performance of selected (left) and rejected (right) policies for the pole-balancing benchmark using observations from 50 episodes. Each curve corresponds to the selection/rejection based on a specific value function (directly from NFQ or generated with an FPE variant). When selection is performed, the very left points give the average reward of the policy considered best (according to the respective selection method); for rejection this corresponds to the policy considered worst. In both cases the very right points of the plots give the average performance of all policies in the set, as in that case all are selected or none rejected, respectively.

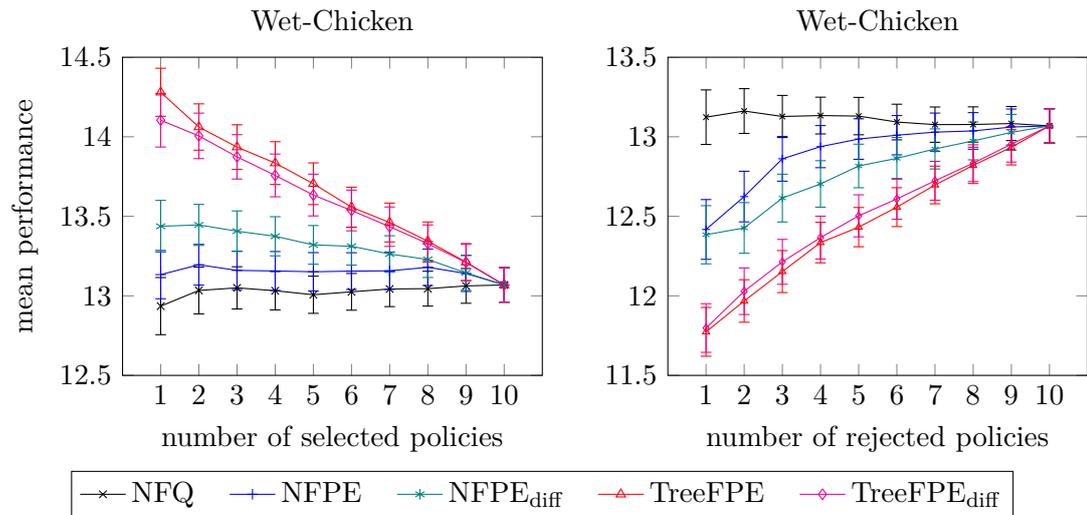
combination with neural FPE becomes obvious (marked green and blue, respectively).

Figure 6.3 shows the results for the wet-chicken domain. Here ensembles with a size of just 10 were used. Again, using the  $Q$ -function from the original NFQ runs it is not possible to distinguish between good and poor policies. Using neural FPE, the situation becomes somewhat better, especially recognizing poor policies becomes possible; the selection, however, does not work as well. Using a different dataset leads to an improvement for recognizing both, good as well as poor policies. But for this benchmark, FPE based on extra-trees works by far best. Interestingly, the extra-trees approach does not need a different dataset to work well. While a different dataset tends to improve the result even further, the tree approach is so different from a neural-network-based FQI that different data does not change the situation much. This is also true for the pole-balancing experiments.

Though not in detail reported here, experiments using extra-tree FPE to evaluate a policy generated by extra-tree FQI resulted in less correlation between the value function and the true performance. Using a different function approximator would probably in this case help as well, i.e., using neural FPE to evaluate policies generated by tree FQI.



**Figure 6.2:** Mean performance of selected (left) and rejected (right) policies for the pole-balancing benchmark using observations from 100 episodes. See the description of Figure 6.1 for more details.



**Figure 6.3:** Mean performance of selected (left) and rejected (right) policies for the wet-chicken benchmark. See the description of Figure 6.1 for more details.

## 6.6 Weighted Ensembles

Instead of using the policy quality measure to select or reject certain policies, one can also derive a weighting from it and use that for weighted majority voting. In the following, the different  $\hat{J}^{\mu, S_0}$  estimates are used as a basis for a weighting for an ensemble. For each ensemble member, its weight  $w_i$  is determined according

	Pole-Balancing		Wet-Chicken
	50 episodes	100 episodes	
<b>Single</b>	0.222 ± 0.008	0.220 ± 0.008	12.9 ± 0.1
<b>Equal</b>	0.287 ± 0.006	0.280 ± 0.005	13.2 ± 0.1
<b>NFPE</b>	0.290 ± 0.006	0.287 ± 0.005	13.2 ± 0.1
<b>TreeFPE</b>	0.296 ± 0.006	0.301 ± 0.004	13.7 ± 0.1
<b>NFPE<sub>diff</sub></b>	0.299 ± 0.006	0.292 ± 0.004	13.5 ± 0.1
<b>TreeFPE<sub>diff</sub></b>	0.296 ± 0.006	0.298 ± 0.004	13.8 ± 0.1
<b>Neg. NFPE</b>	0.244 ± 0.007	0.226 ± 0.007	13.0 ± 0.1
<b>Neg. TreeFPE</b>	0.260 ± 0.007	0.236 ± 0.007	12.3 ± 0.1
<b>Neg. NFPE<sub>diff</sub></b>	0.230 ± 0.008	0.212 ± 0.008	12.8 ± 0.1
<b>Neg. TreeFPE<sub>diff</sub></b>	0.261 ± 0.007	0.237 ± 0.006	12.4 ± 0.1
<b>Uncorrelated</b>	0.285 ± 0.006	0.278 ± 0.005	13.2 ± 0.1
<b>MP NFPE</b>	0.255 ± 0.005	0.273 ± 0.007	13.1 ± 0.1
<b>MP TreeFPE</b>	0.273 ± 0.006	0.294 ± 0.006	14.1 ± 0.1
<b>MP NFPE<sub>diff</sub></b>	0.258 ± 0.008	0.273 ± 0.007	13.4 ± 0.1
<b>MP TreeFPE<sub>diff</sub></b>	0.272 ± 0.007	0.298 ± 0.006	14.1 ± 0.1

**Table 6.3:** Performances of differently weighted ensembles. *Equal* denotes equal weighting of all members; *NFPE* and *TreeFPE* denote ensembles weighted using weights derived by neural and extra-tree FPE, respectively; for *Neg. NFPE* and *Neg. TreeFPE* the same weights were used, only negated; *Uncorrelated* denotes uncorrelated weights. *MP* denotes the most preferable policy, i.e., the policy with the greatest weights. The subscripted *diff* denotes usage of a different dataset of FPE.

to

$$w_i := \frac{\hat{J}^{\mu, S_0}(\pi_i) - \min_j \hat{J}^{\mu, S_0}(\pi_j)}{\max_j \hat{J}^{\mu, S_0}(\pi_j) - \min_j \hat{J}^{\mu, S_0}(\pi_j)}, \quad (6.5)$$

thus scaling the weights between 0 and 1. By increasing the influence of good and decreasing the influence of poor ensemble members, one may hope to increase the overall performance of the ensemble. If, on the other hand, the weighting is systematically wrong, i.e., good policies are weighted lower than poor policies, a weighted ensemble should perform worse than an equally weighted one. Uncorrelated weights should have no impact on an ensemble’s performance.

Table 6.3 contains the results of the weighting experiments for the pole-balancing benchmark using 50 and 100 episodes as well as the wet-chicken benchmark. As a baseline, the performances of equally weighted ensembles (second row) are given. Additionally, the average reward of a single network policy (without ensemble) is shown (first row). The next two rows give the performances of ensembles weighted using  $\hat{J}_{\text{NFPE}}^{S_0}$  and  $\hat{J}_{\text{TreeFPE}}^{S_0}$ , respectively. Here, the weighting consistently improves

Pole-Balancing		
	50 episodes	100 episodes
<b>Single</b>	$0.222 \pm 0.008$	$0.220 \pm 0.008$
<b>Most agreeable</b>	$0.276 \pm 0.007$	$0.278 \pm 0.007$
<b>MP NFPE</b>	$0.255 \pm 0.007$	$0.273 \pm 0.007$
<b>MP TreeFPE</b>	$0.273 \pm 0.007$	$0.294 \pm 0.005$
<b>MP NFPE<sub>diff</sub></b>	$0.258 \pm 0.007$	$0.273 \pm 0.007$
<b>MP TreeFPE<sub>diff</sub></b>	$0.272 \pm 0.007$	$0.298 \pm 0.006$

**Table 6.4:** Performances of policies from the “most agreeable” approach from the previous chapter and “most preferable” policies (i.e., the policy considered best by the respective evaluation method).

the performance, especially when using extra-tree-based FPE. When doing the same weighting using a different dataset (rows 5 and 6), the NFPE-based results can further be improved, while the results of TreeFPE remain approximately the same. This is in agreement with the plots from the previous section, which always show an improvement for NFPE with different data over the same dataset, whereas the results of TreeFPE for the same and a different dataset are very similar. Next, the effect of deliberately bad weights is shown by negating  $\hat{J}_{\text{NFPE}}^{S_0}$  and  $\hat{J}_{\text{TreeFPE}}^{S_0}$ ; the results are given in rows 7 and 8. As expected, the performances decrease when deliberately choosing bad weights. The better the correlation between  $\hat{J}^{S_0}$  and the true performance, the more drastic the performance decrease when negating the corresponding weights. Rows 9 and 10 show the same for different datasets. Again, NFPE’s performance is affected (better correlation, therefore worse performance when negating weights), while for TreeNFPE no difference can be observed. *Uncorrelated* gives the results of using uncorrelated weights. As expected, the performances of ensembles weighted with uncorrelated weights tend to be lower because of the smaller effective ensemble size (since all weights except one are smaller than 1, the effective ensemble size is reduced in comparison to unweighted ensembles). However, this is the only effect of uncorrelated weights. Lastly, results of the *most preferable* policy (*MP*) are given, where the single policy with the greatest weight was selected. Unsurprisingly, the better the correlation between quality measure and the true performance, the better the selection of a single policy. Note that the numbers in the table correspond exactly to the leftmost points of the left-hand plots in the previous section. Pole-balancing profits much more from ensembles than wet-chicken, where the best results are achieved by single policies. This is probably due to the highly stochastic nature of the wet-chicken benchmark and the fact that a policy usually has two “switching points”: at the beginning of the river, *drift* is the preferred action; after the first “switching point”, *hold* becomes the best action, after the second action *back*. Mixing policies with different “switching points” apparently

does not work as well. Another explanation is the superiority of TreeFPE, which allows to select a single NFQ policy that performs better than an ensemble.

Table 6.4 compares “most preferable” policies to those identified by the “most agreeable” approach from the previous chapter, where the policy is chosen that most often is among the majority for the states appearing in the observation dataset (the numbers for MP are repeated from Table 6.3). It shows that both approaches perform comparably. In the setting using observations from 50 episodes, the “most agreeable” approach has advantages over NFPE-based “most preferable” policy selection and gives similar results compared to the TreeFPE-based variant. In the setting using observations from 100 episodes, an advantage for TreeFPE-based policy selection over the “most agreeable” approach can be observed. With more data available, the approach based on direct policy assessment more clearly identifies good policies. However, if due to constraints at runtime (policy execution) only a single policy can be used, it seems questionable whether the additional effort of running FPE is worthwhile, given that the much cheaper process of determining the “most agreeable” policy leads to a comparable result. If during runtime an ensemble is to be used, though, determining and using a weighting does make sense, as it leads to a further performance improvement.

## 6.7 Summary

In the present chapter methods for policy assessment in continuous domains were discussed. Starting from the natural approach of using the value function, it was argued why an uncertainty estimate comparable to self-assessment in discrete domains cannot be obtained in a similar fashion. The correlation coefficient was used as a measure for suitability of value functions determined in different ways. The correlation between the value function generated during policy generation with NFQ and the true performance is often almost zero. If, however, a separate policy evaluation step is performed, the correlation can be improved considerably. To this end the fitted policy evaluation (FPE) approach was introduced. Moreover, it was argued that by employing a different function approximator and/or using a different observation set, the correlation can be increased further. It followed an evaluation of the approaches for policy selection and rejection. In addition, the policy performance measures were used to construct weighted ensembles, showing that the ensemble performance can be improved through weighting. Also the “most agreeable” policy selection approach from the previous chapter was compared with the weighting-based “most preferable” one. The “most agreeable” policy is the one that is most often among the majority (in majority voting) on the observation set, while the “most preferable” policy is that with the highest weight. It turned out that the “most agreeable” approach performs quite comparably to the “most preferable” one.

In addition to the applications evaluated here (policy selection/rejection, ensem-

---

ble weighting) others are thinkable. In a practical application, one could define a minimum value that a policy must reach during evaluation in order to be put into action. One can also use it to switch to a new policy only when a considerable performance gain is to be expected. This can be assessed by comparing evaluation results of the currently used and the candidate policy.



# 7

## Autonomous Control in Changing Environments

This chapter combines the ideas from the previous two chapters for approaches to deal with changing environments. As a changing environment this thesis considers a problem whose characteristics slightly vary, while the general structure remains unchanged. For example, in a gas turbine those changing characteristics could be fuel quality, ambient parameters not included in the state representation, or changing characteristics of the turbine itself due to wear. Such a changing environment violates the Markov property. Therefore, the theoretically sound solution is to include all parameters describing the change in the state space. However, not only is it often hard, if not impossible, to observe such changes directly, but a state space with greater dimensionality usually requires the collection of more observations to generate a near-optimal policy. Instead, if the change in characteristics occurs slowly enough, one can assume the environment to be static for a limited timeframe. This assumption makes it possible to generate a new policy regularly, using data from a new, recent timeframe. Exactly this approach is used in this chapter.

Two settings of changing environments are considered. In the first, a pool of policies and an environment are given. All policies in the pool are known to work reasonably for the environment. They were generated using observations from environments of a certain class but with different parameterizations. It is unknown what the parameterization of the current environment is. The task then is to select policies from the pool that are best suited for the given environment.

In the second setting, a changing environment is considered and the aim is to generate policies in a way that at each point in time a policy is available that controls the environment reasonably well. The traditional solution is to regularly generate a new policy using the last  $n$  observations. Here, an approach is proposed using an evolving ensemble, where new policies are generated regularly as in the traditional approach, but instead of a single policy an ensemble is used. The ensemble *evolves* by adding and removing policies, optionally based on policy evaluation.

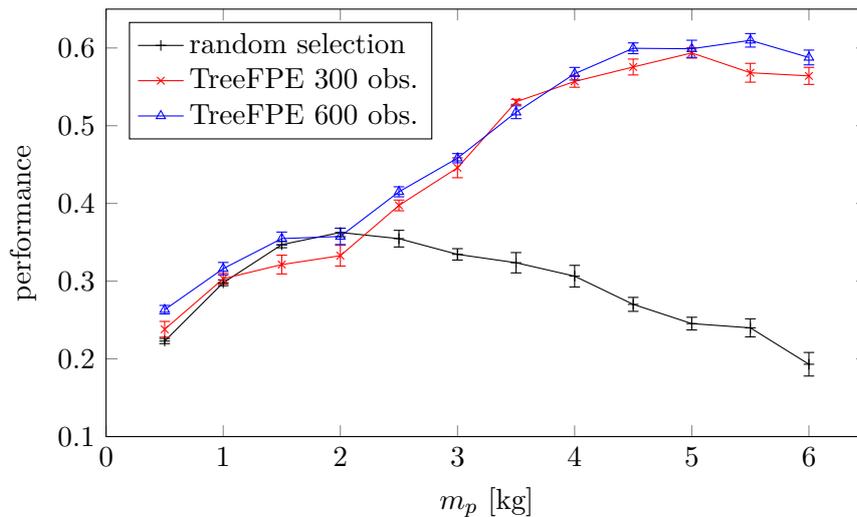
## 7.1 Policy Selection in Changing Environments

Consider the following setting: Given a set of policies, determine a weighting for the policy set or select a number of policies for a new Markov decision process (MDP) that is similar to the one the policies were determined for, but has a somewhat different parameterization. In a real-world scenario, for the given policies an expert might have assured that they work reasonably for the given system to be controlled. Changing characteristics of the system due to, e.g., environmental conditions (temperature, air pressure, etc.) or wear, could be regarded as different parameterizations. A possible solution for this setting could be as follows:

1. Select a number  $n$  of policies from the set randomly to form an ensemble.
2. Let the ensemble run for a number of steps and save the observations. Since all policies were assessed by an expert beforehand, the ensemble's performance will be reasonable, though in general not the best achievable (due to the lack of adaptation to the current parameterization).
3. Use the observations from the previous step to evaluate each policy from the set for the given parameterization (represented by the observations). From the policy evaluations a weighting can be obtained.
4. Use the weights to select the presumably best  $n$  policies from the set to form a new ensemble.
5. Use the new ensemble to control the system.

The above procedure can be repeated regularly or every time the average performance drops below a threshold, indicating a changed parameterization. Moreover, some form of bootstrapping can be applied, where first a weighting is determined using only few observations. With the ensemble from the first weighting more observations are generated that again can be used to obtain a better weighting. This way the amount of time the ensemble of randomly selected policies is used could be minimized.

This approach is evaluated on the pole-balancing benchmark. Changed parameterizations are realized by varying the mass of the pole  $m_p \in \{0.5, 1, 1.5, \dots, 6\}$  kg. For each parameterization, five datasets were generated, each with observations from 100 episodes, using random exploration. For each dataset, five neural fitted  $Q$ -iteration (NFQ) runs were performed (as described in Chapter 5) to obtain five policies, totaling in 25 policies per variation and 300 policies for all variations. Those 300 policies comprise the *pool* of policies assumed to work reasonably well for all parameterizations. To test the method for a particular parameterization,  $n$  policies were selected randomly from the pool, forming an ensemble, which was run for 300 or 600 steps, respectively. With the resulting observations, all 300



**Figure 7.1:** Performance of equally weighted ensembles selected randomly and using TreeFPE based on 300 and 600 observations for different variations of the pole-balancing benchmark (different pole mass  $m_p$ ). For all ensembles 20 members were used.

policies were evaluated using TreeFPE, resulting in  $\hat{J}_{\text{TreeFPE}}^{S_0}(\pi_i), i = 1, \dots, 300$ , which were used to order the policies and then select the presumably best  $n$  from the total 300 to form an ensemble. The resulting ensembles were evaluated (100 trials with at most 3,000 steps each), as a baseline the performance of ensembles with randomly selected members was determined as well.

Figure 7.1 shows the performances of equally weighted ensembles for different parameterizations of the benchmark (varying pole mass  $m_p$ ). Black indicates the results of ensembles with randomly selected members. Apparently, with  $m_p \in [1, 5]$  kg the problem is easier than with very small or large weights. This somewhat counter-intuitive result can be explained as follows: When the pole is quite light, a few wrong actions can already lead to failure. If it is heavier, it has more inertia, and a single action has less impact. On the other hand, if it becomes too heavy, it becomes increasingly difficult to erect a pole that is not standing straight, and a policy must use the right actions as often as possible to use all the available force in the right direction.

The pool contains policies trained with observations from poles with different masses. When selecting policies randomly from the pool (black curve), one can do quite well for  $m_p \leq 2.5$ , since most policies are suitable for those parameterizations. However, to excel at parameterizations with higher pole mass, a random selection is insufficient. In contrast, the ensembles built from policies considered best by TreeFPE by far outperform the ones based on random selection. TreeFPE was run with 300 observations (red) and 600 observations (blue), but the evaluation quality benefits only little from the extra observations.

## 7.2 The Evolving Ensemble

In this section we discuss an approach to use ensembles and policy evaluation in an on-line system able to learn a policy from scratch, control the system using that policy, and also update the policy regularly to react to changing characteristics.

### 7.2.1 Idea

The basic idea again comprises a policy pool. Some or all policies from that policy pool form the ensemble that constitutes the policy. Whenever a certain number of new observations has been collected, they are used to generate one or more new policies. Those policies are then added to the pool, and all policies in the pool are re-evaluated using recent data. The results of this re-evaluation can then be used to update the ensemble and possibly remove policies from the pool that—according to the evaluation—perform insufficiently.

The naïve approach of using a single policy and regularly replacing it with a new one trained from recent observations has a number of shortcomings:

- As we have seen in Chapter 5, a policy resulting from a single NFQ run might perform insufficiently. It is advisable to use ensembles to weaken this problem.
- Datasets often vary in quality, leading to policies of varying quality. Even when using ensembles, a poor dataset will lead to poor policies.
- Replacing the entire ensemble might lead to abruptly changing policies. In general, gradually changing policies are preferred.

The evolving ensemble addresses these problems mainly through ensemble usage. First, an ensemble reduces the risk of picking a single poor policy. Since the majority of policies performs reasonably, an ensemble will most likely perform reasonably as well. Second, by replacing only a part of the policy pool and not all policies at once, the influence of a single poor dataset is decreased. This also ensures a gradual change of the ensemble policy, because only a part of the ensemble can be changed from one step to the next. Third, by using policy evaluation to decide which policies are kept and which are removed from the pool, the influence of poor policies is further reduced.

In detail, the evolving ensemble approach works as follows:

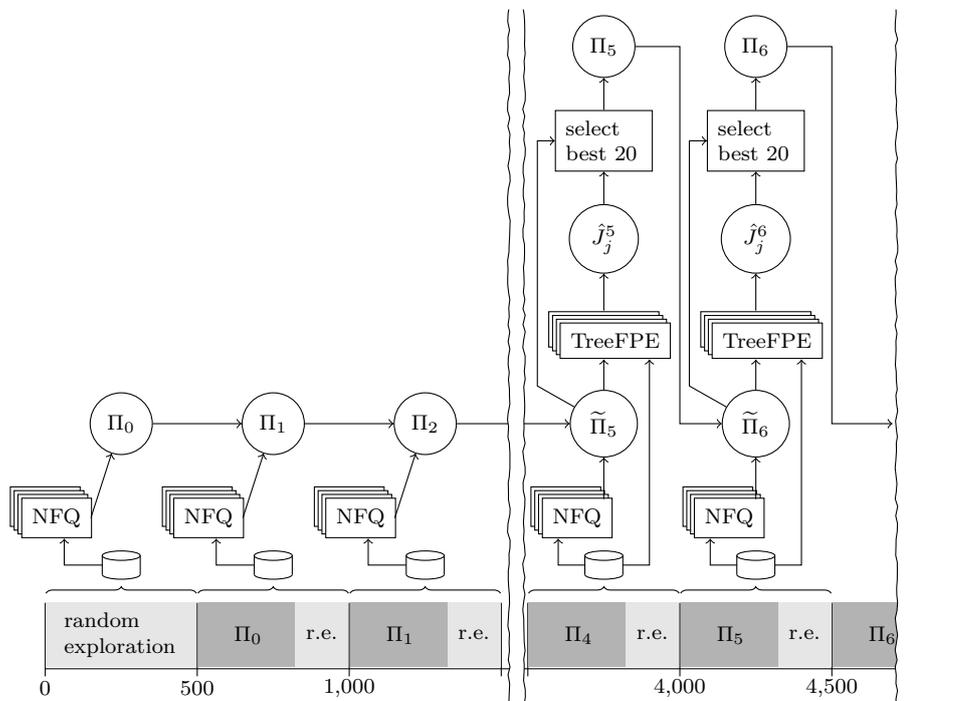
1. Initialization. Starting with some dataset (e.g., generated by random exploration), determine  $K$  policies. These policies form the initial policy pool  $\Pi_0$ . An ensemble containing all (or a random subset of) policies from  $\Pi_0$  forms the first ensemble policy  $\pi_0^e$ . A counter  $i$  is initialized  $i := 0$ .

2. Policy Execution.  $\pi_i^e$  is used to control the system for  $N$  steps, thereby generating  $N$  observations.
3. Policy Generation. The  $N$  most recent observations are used to determine  $K$  new policies, which are added to the policy pool  $\Pi_i$ .
4. Policy Evaluation. All policies  $\pi_j \in \Pi_i$  are evaluated with the  $N$  most recent observations (e.g., using TreeFPE). The result for a policy  $\pi_j$  is stored as  $\hat{J}_j$ .
5. Policy Pruning. If the size of the policy pool is greater than some maximum capacity  $M$ , i.e.  $|\Pi_i| > M$ , the worst  $|\Pi_i| - M$  policies (according to  $\hat{J}_j$ ) are removed from  $\Pi_i$ .
6. Ensemble Formation. A new ensemble policy  $\pi_{i+1}^e$  is formed using the best  $L$  policies from  $\Pi_i$  (according to  $\hat{J}_j$ ).
7. Loop. The counter  $i$  is increased and it is continued with step 2.

When learning new policies from observations generated by previous, already near-optimal policies, the problem of imbalanced data arises. A near-optimal policy will visit only certain areas of the state space. This can lead to an under-representation of observations of other areas. Furthermore, such a policy will in a given small area of the state space always choose the same action. As a consequence, a policy generated from those observations might be able to control the system well in a preferred area of the state space, but at the same time be unable to properly move the system there. Although the problem’s severity will be different depending on the environment and the generalization capabilities of the function approximator, in general it must be considered and taken care of. Possible solutions include weighting or resampling of observations (Abtahi and Fasel (2011) report good results by subsampling explorational data, where most observations are from regions outside the target area). For an on-line setting of batch-mode reinforcement learning (RL), as discussed here, it is also thinkable to remember all observations and resample before learning a policy. The resampling would then make sure that all areas of the state space are represented equally well. However, when dealing with changing environments the age of an observation becomes important as well—the older an observation, the less trustworthy it is. It is therefore unavoidable to include some sort of exploration to circumvent the problem of imbalanced data or even unrepresented regions of the state space.

### 7.2.2 Experiments

The approach is first evaluated on the pole-balancing benchmark. Analogously to the experiments in the previous section, the mass of the pole is varied. This time, however, we are not presented a concrete parameterization and have to



**Figure 7.2:** Setup of the evolving ensemble experiment for the pole-balancing environment. The axis depicts the timestep. Shown are the first 1,500 timesteps (left) and timesteps 3,500–4,500 (right). For the first 500 timesteps, random exploration is performed. The resulting observations are used for four NFQ runs, resulting in four policies, constituting the first ensemble pool  $\Pi_0$ . The next  $500 \cdot 4/5 = 400$  steps ensemble  $\Pi_0$  is executed, again saving the observations. The remaining  $500 \cdot 1/5 = 100$  steps random exploration is performed. The observations from both, ensemble policy execution and random policy are used to determine another four policies, which are then added to the pool. Now,  $|\Pi_1| = 8$ . This continues until the size of the pool exceeds the maximum size of  $M = 20$ . After timestep 3,500 this happens for the first time. Hence,  $\tilde{\Pi}_5$  contains 24 policies. For all 24 policies, policy evaluation, namely TreeFPE, is performed, using the observations from the recent 500 observations. The resulting quality measures  $\hat{J}_j^5$  are then used to select the presumably best 20 policies, forming  $\Pi_5$ , which is then used as the ensemble. After the next 500 steps again four policies are generated and added to  $\Pi_5$ , which then again is too large, so the presumably best 20 policies are selected, etc.

select suitable policies from a given policy pool. Instead, we are faced with a single environment that changes slowly. While the environment changes, we want to adapt the executed control policy to accommodate for the changes as good as possible.

To implement a gradual mass change, the mass  $m_p$  is defined to be time dependent according to

$$m_p^0 := 0.5 \tag{7.1}$$

$$m_p^t := m_p^0 + 5.5 \sin\left(\frac{\pi}{2 \cdot 500 \cdot 100} t\right), \tag{7.2}$$

where  $m_p^0$  is the initial mass,  $\pi$  the mathematical constant (expressing the ratio of the circumference of a circle to its diameter) and  $t$  the current timestep. With the used frequency a quarter period (and thus the maximum mass) is reached after  $500 \cdot 100$  timesteps.

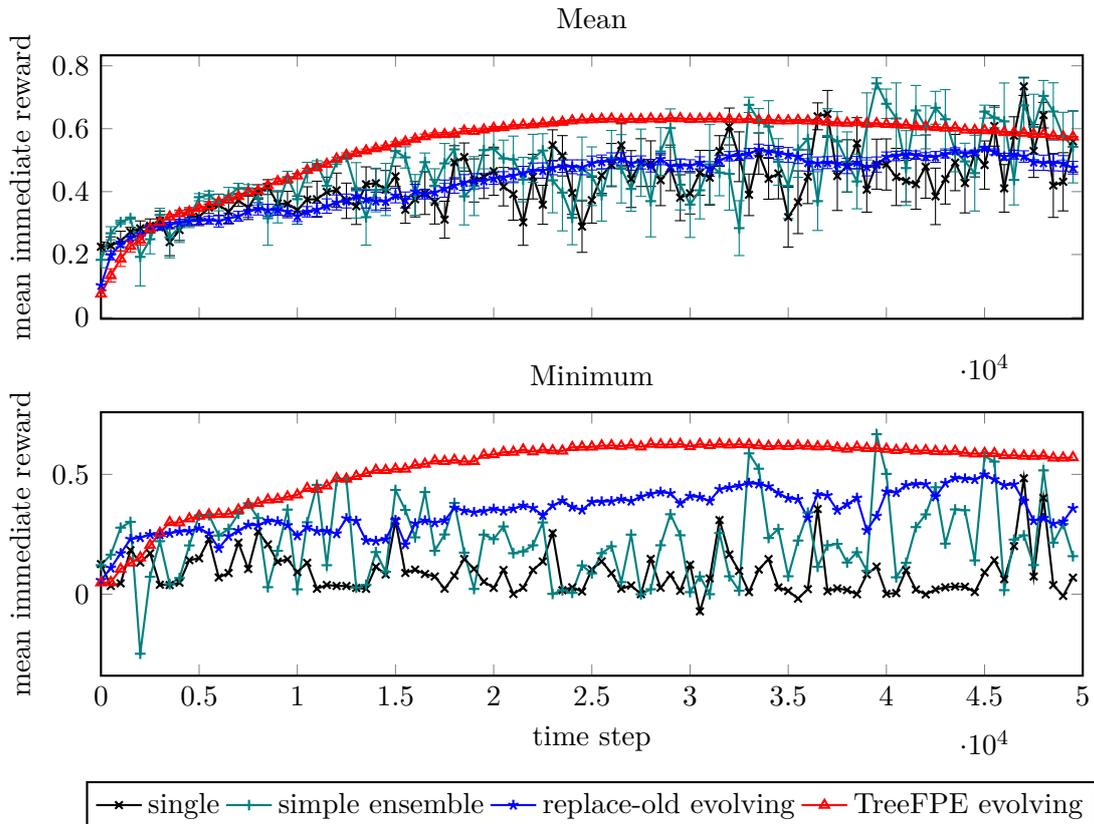
Every 500 timesteps the recent 500 observations are used to generate four new policies, i.e.,  $N := 500$  and  $K := 4$ . For policy evaluation TreeFPE is used. The maximum policy pool size is set  $M := 20$ . The complete policy pool is used as ensemble, thus  $L := M = 20$ .

To deal with the problem of imbalanced data, the ensemble policy is used for only  $\frac{4}{5}N$  steps. The remaining  $\frac{1}{5}N$  steps random exploration is used. Obviously, for a real-world application another solution would be required, but here the focus is on the evolving ensemble approach. The problem of imbalanced data occurs as well with static environments that do not change at all.

Figure 7.2 illustrates the experimental setup.

The evolving ensemble is compared with three less sophisticated approaches. First, the traditional approach is used, where a single new policy is generated using the recent 500 observations (*single policy*). The rest of the setup is identical, i.e., the policy is executed for  $\frac{4}{5}N$  and random exploration is used for the remaining  $\frac{1}{5}N$  timesteps. Second, not a single policy, but an ensemble of four policies (derived from the same dataset) is used (*simple ensemble*). Third, the evolving ensemble approach without the policy evaluation step is considered. Instead of performing policy evaluation, once the ensemble has become too large, policies are removed starting with the oldest policy (*remove-old evolving ensemble*).

For each approach 10 independent trials were performed as described above. Each trial led to 100 policies, one every 500 timesteps. Note that in this context a “policy” can as well be an ensemble composed of up to 20 single policies. To measure the quality of a policy, its mean immediate reward was determined by running it on an instance of the pole-balancing problem with a pole mass corresponding to the respective timestep. Figure 7.3 shows the results. The upper plot shows the means over the 10 trials, the bottom plot shows the minimum over



**Figure 7.3:** Results from the evolving ensemble experiments using the pole-balancing benchmark. The upper plot shows mean values over the performed trials, the bottom plot shows the minimum over all trials.

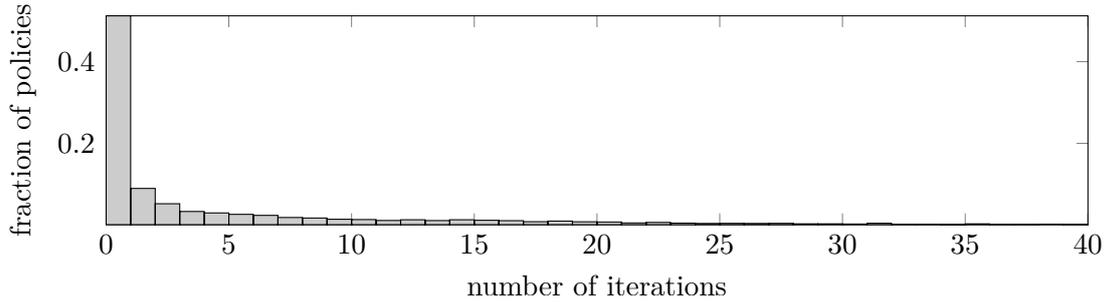
all trials for each timestep. As we have already seen when varying the mass of the pole at the beginning of this chapter, the pole-balancing problem becomes easier with increasing pole mass (due to the increased inertia) up to certain point. The different approaches exhibit notable differences in their ability to exploit this simplification.

Single policies (black, ‘×’ marks), where just a single policy is trained using the recent 500 observations, and simple ensembles (green, ‘+’ marks), which are composed of four policies trained on the recent 500 observations, show mixed results. While occasionally good performance is achieved, the policy quality often changes rapidly from one update to the next, i.e., after every 500 timesteps. Moreover, when looking at the minimum results of all trials, single policies and ensembles turn out worse than the other approaches. Although it is not that obvious from the plots, the simple ensembles perform better on average than single policies (0.47 for ensembles compared to 0.42 for single policies), especially when it comes to the minimum results (0.22 for ensembles and 0.09 for single policies).

The evolving ensemble approach removing the oldest policies, once the ensemble becomes too large (blue, ‘★’ symbol), delivers a more consistent performance. This is in part due to the greater ensemble size of 20, and in part due to the smaller change from one update to the other: of the 20 policies only four change in an update. When looking at the average of all timesteps, with 0.43 it performs worse than the simple ensemble approach and only slightly better than single policies, which is somewhat surprising. When it comes to the minimum, however, it can clearly outperform the two simpler approaches (with an average of 0.34). The smoothing effect of the remove-old evolving ensemble approach keeps it from taking full advantage of the very best policies, which in some timesteps are generated. At the same time it keeps it from incurring the poor performance of the occasional poor policies. This latter feature is what makes this approach attractive for autonomous control, where we look for consistent reasonable performance.

Finally, the evolving ensemble approach with TreeFPE-based policy assessment (red, ‘△’ marks) is superior to all others in this setting. Except for the first timesteps it consistently delivers high performance. This is true over all 10 trials, since even the minimum over all trials is not much worse than the mean (the maximum difference between the mean and the minimum is 0.10, the mean difference 0.02). It is also reflected in the low uncertainty of the mean (small error-bars). Since TreeFPE works well for determining a policy’s performance here, it becomes possible to not just remove the oldest policies, but those that most probably are unsuitable—either because the environment has changed too much since their generation, or because they were poor policies in the first place.

Figure 7.4 shows a histogram of the number of timesteps policies are used before they are discarded. The histogram shows this in terms of iterations, i.e., the time between updates of the ensemble (generation of new policies and assessment of all policies using recent observations, removal of assumingly poor policies). As can be seen from the figure, about half the policies are not executed even once; they are generated, thereafter assessed using the same dataset (but with TreeFPE a different method), and found to be worse than other candidate policies. This is in part due to the tendency of NFQ to occasionally produce poor policies (as discussed in Chapter 5). Moreover, candidate policies not only have to “compete” with policies generated from the same dataset; also the 20 policies already present in the ensemble are candidate policies. Since they potentially have been assessed several times before and always have been among the best 20 policies (otherwise they were not part of the ensemble anymore), chances are they actually *are* good policies. At the same time, since the environment is changing, even the best policies will become unsuitable eventually and are removed from the ensemble. With an increasing number of iterations, the fraction of policies decreases, which is not surprising. Policies are expected to perform best for the environment that generated the observations they were learned with. Only few policies are so universal that they fit a large timeframe.



**Figure 7.4:** Histogram of the duration of policy usage. An iteration here is the time between two updates, i.e., 500 timesteps. Half the policies are discarded before being executed even once (large bar for 0 iterations). As expected, the fraction of policies decreases with increasing policy age (number of iterations the policy is part of the ensemble).

The comparison of simple 4-policy ensembles on the one hand and ensembles with 20 members in the evolving ensemble case on the other hand might seem unfair. The amount of NFQ runs involved is identical, though. The important difference lies in the way policies are used. In the simple ensemble approach a policy is used only for a single ensemble; the evolving ensemble uses policies multiple times and takes advantage of the fact that the environment changes only slowly. The approach with assessment-based removal admittedly adds the cost of running TreeFPE for all 20 policies every 500 observations. However, looking at the results this is worth the effort. In the end, it means using (comparatively cheap) computational power to extract more information out of (comparatively expensive) observations of the environment.

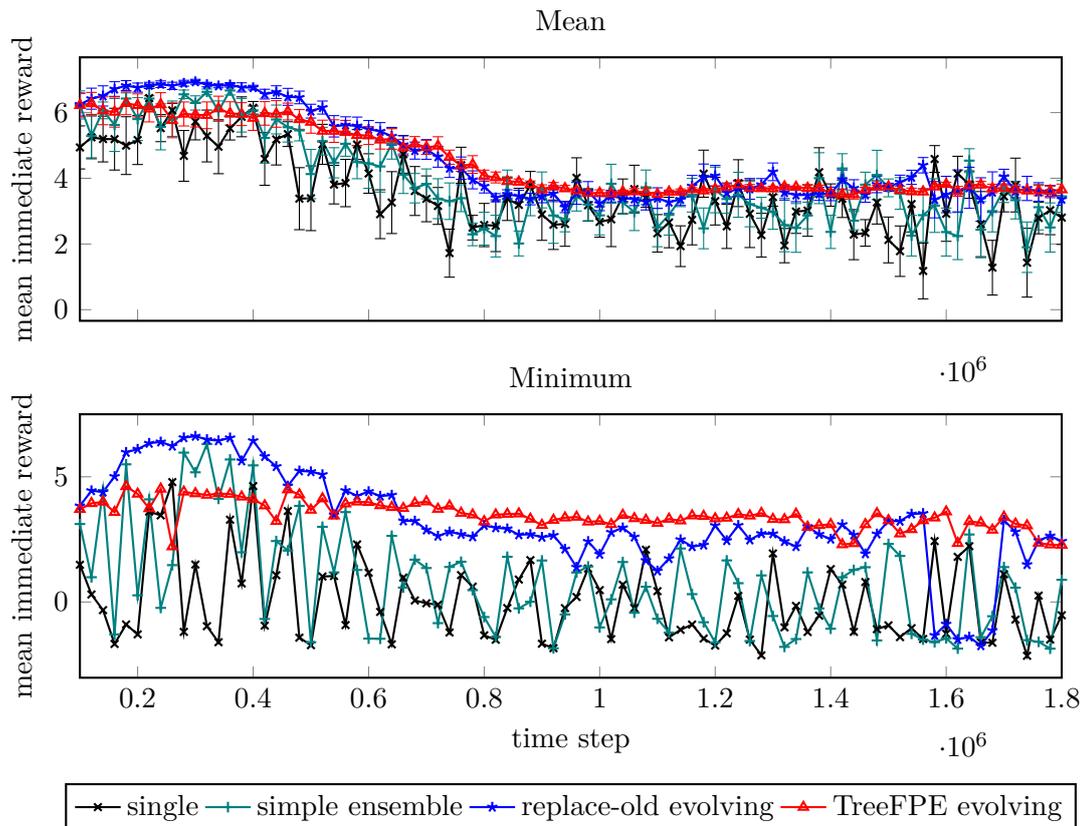
The evolving ensemble approach was also evaluated on the more difficult cart-pole benchmark as described in Chapter 5, Section 5.4.1. Since one needs much more observations to have a chance at deriving reasonable policies, new policies are learned not every 500, but every 20,000 observations. To change the environment’s characteristics, the pole’s length  $l_p$  is changed according to

$$l_p^0 := 0.25 \quad (7.3)$$

$$l_p^t := l_p^0 + 0.5 \sin\left(\frac{\pi}{2 \cdot 1.8 \cdot 10^6} t\right), \quad (7.4)$$

where  $l_p^0$  is the initial length, again  $\pi$  the mathematical constant and  $t$  the current timestep. With the used frequency a quarter period is reached after  $1.8 \cdot 10^6$  timesteps, thus the maximum length is reached at  $t = 1.8 \cdot 10^6$  with  $l_p^t = 0.25 + 0.5 = 0.75$ .

Each dataset is used to generate four policies, i.e.,  $K := 4$ . Every 20,000 observations new policies are generated, i.e.,  $N := 20,000$ . For policy evaluation TreeFPE is used. The maximum policy pool size is set  $M := 20$ . The complete



**Figure 7.5:** Results from the evolving ensemble experiments using the cart-pole benchmark. The upper plot shows mean values over the performed trials, the bottom plot shows the minimum over all trials.

policy pool is used for an ensemble, thus  $L := M = 20$ . So only  $N$ , the number of observations after which new policies are generated, is different from the previous setup using the pole-balancing benchmark.

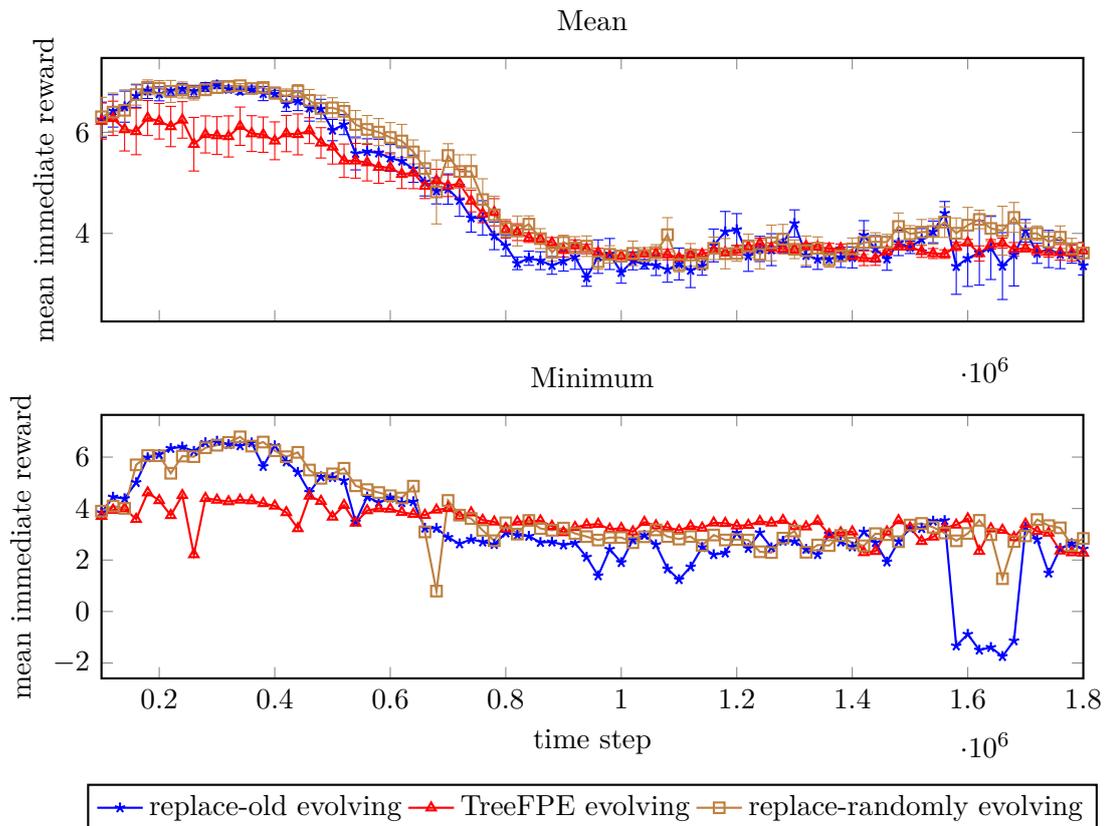
To deal with the problem of imbalanced data, again random exploration is used, here for  $\frac{1}{3}N$  timesteps for each update (to account for the larger state space).

As previously with the pole-balancing benchmark, four approaches are compared: a single policy learned from the recent 20,000 observations, an ensemble of four policies learned from the recent 20,000 observations, an evolving ensemble that adds four policies every 20,000 timesteps (again learned from the recent 20,000 observations) and removes the oldest policies once the ensemble size has reached its limit, and an evolving ensemble that uses TreeFPE to decide which policies to remove. Again, 10 trials were performed. Figure 7.5 shows the results. While the task of pole-balancing becomes easier with increasing pole mass, the cart-pole problem appears to become harder with increasing length of the pole— all approaches show some decline in performance with increasing pole length. However, differences between the approaches can be observed again.

Single policies (black, ‘×’ marks) perform worst, simple ensembles containing four members (green, ‘+’ marks) are slightly better. The best results, especially in terms of the minimum mean immediate rewards (bottom plot) are achieved by the evolving ensemble approaches. This comes as no surprise—after all, the evolving ensembles make more effective use of the generated policies by using them multiple times. Between the two evolving ensemble approaches there are interesting differences, though. First, in the first third of the total timeframe (approx. timestep 0-800,000) the evolving ensemble without assessment (and instead removal of the oldest policies) performs better than the one using assessment; this is true for the mean as well as the minimum over all trials. Apparently, the policy assessment done by TreeFPE is often wrong, leading to a selection of poor policies. The second interesting observation occurs around timestep 1,600,000: the minimum of the evolving ensemble with removal of old policies drops for several timesteps below zero; at the same time, the evolving ensemble with assessment can maintain its performance. In the following, we will investigate the reasons for both observations.

The TreeFPE-based policy assessment is supposed to allow for keeping good and discarding poor policies. We therefore expect it to be a good measure for the quality of single policies. As done before, it is useful to consider the correlation coefficient between the results from TreeFPE and average rewards from execution on the actual problem: over all policy evaluations the correlation is  $-0.02 \pm 0.04$ . Given this correlation, doing TreeFPE seems useless in this setting—one could as well select policies at random. On the other hand, the correlation between all evaluations of the run and the respective policies’ true performance is not exactly what is interesting. Instead, one should investigate the policy pool of each iteration individually. In each iteration, TreeFPE is performed for each policy currently in the pool using the same observations; the resulting ranking is used to select the assumingly best 20 policies (and discard the poorest four). Therefore, the correlation of the 24 policies of each of those sets and their true performance is what is really interesting. Determining the corresponding numbers, we get a correlation of  $-0.17 \pm 0.02$  over all trials. This result is even worse—it means that systematically the wrong policies are selected, though the effect is weak as the absolute value of the correlation is rather small. This explains the poor performance during the first third of the experiment. Given the relative performance in the first third and the rest of the experiment, one would expect the negative correlation to be more severe during the first third. However, a systematic change in correlation over time was not found.

Now consider the drop of the minimum performance of the evolving ensemble with removal of old policies around timestep 1,600,000. In this case, the ensemble contains a number of poor policies; nonetheless, majority voting still leads to a high quality of the ensemble policy. In the next iteration, however, two of the four new ensemble members are poor, while four good policies are removed. After that the number of good members is not sufficient to keep up the good ensemble performance. The next two updates cannot significantly change the situation,



**Figure 7.6:** Results from the randomly replacing evolving ensemble. The results from the ensemble replacing old policies and the TreeFPE-based evolving ensemble are shown for comparison; they are identical to those in figure 7.5.

as they add one or more poor policies themselves. Moreover, the poor policies added in previous iterations are still present. When finally in the next iteration four good policies are added, it still takes two iterations for the performance to resume an acceptable level, because a sufficient number of poor policies needs to be shifted out.

One would expect that this problem could be avoided by using policy evaluation to select which policies to keep and which to discard. An according experiment was performed by using the same single policies in combination with TreeFPE: instead of in each iteration adding four new and discarding the oldest four policies, the decision was based on TreeFPE policy evaluation. Interestingly, this approach discarded a sufficient number of poor policies to prevent the performance drop. This result is somewhat surprising, since the correlation between true policy performance and TreeFPE suggests that the quality measure derived by TreeFPE is not informative in this setting. Indeed, if the correlation is approximately zero, using random numbers as performance measure should lead to the same result. To this end the experiment was repeated with random selection:

when in an iteration the ensemble becomes too large and policies need to be dropped, those policies are selected at random. Figure 7.6 shows the result and compares them with the other two evolving ensemble approaches. Remarkably, the mean performance over all 10 trials of this approach is equivalent to (if not better than) the performance of the other evolving ensemble approaches. Looking at the minimum, the random selection performs comparably to the evolving ensemble dropping the oldest policies in most timesteps, but it can avoid the huge performance drop towards the end. This interesting result can be explained by the fact that the random selection ensemble contains policies from a broader range—it does not always drop the oldest policies, consequently some old policies remain in the ensemble. This weakens the impact of the sequential poor iterations, where each iteration adds a number of poor policies. At the same time, the older a policy, the smaller its probability of not having been dropped previously. Taking the numbers of this experiment, in each iteration 4 of 24 policies need to be dropped, so the probability of being dropped in one iteration is  $4/24 = 1/6$ . From this follows that the probability for a policy that entered the ensemble in iteration  $i$  still being a member in iteration  $i + k$  is  $(1 - \frac{1}{6})^k$ .

### 7.2.3 Conclusions

From the observations with the above experiments the following conclusions can be drawn:

- The evolving ensemble approach in general works well in changing environments. Not only does it deliver a better mean performance than individual ensembles or even single policies, it most importantly improves upon the minimum performance. Instead of potentially drastically changing the policy with each update, it provides for gradual change of the policy and adaptation to the environment. Of course, it is as well applicable if the environment is static and does not change.
- If a means of policy evaluation is available, it can be a great addition to the evolving ensemble approach, since then one can selectively remove poor and keep good policies. The pole-balancing experiment showed that this can lead to a remarkable advantage. On the other hand, if the quality of the policy evaluation is poor and perhaps even negatively correlated to the true performance, including policy evaluation can weaken the ensemble. However, as long as the negative correlation is not extreme, the general robustness introduced by the ensemble is sufficient to still keep its performance above those of simple (non-evolving) ensembles and single policies.
- In the experiments with the cart-pole benchmark there was one trial where the evolving ensemble without assessment showed a severe performance drop. This was mainly due to a number of subsequent ensemble updates

that added poor policies. Further experiments using other selection strategies (instead of simply dropping the oldest policies), but based on the same single policies, showed that also for this trial the performance drop can be avoided by essentially widening the range from which policies are selected and hence making the ensemble more diverse. As an alternative to different selection techniques one could also increase the size of the ensemble; this would also lead to the inclusion of policies from a wider range of timesteps. Yet another alternative lies in decreasing the number of policies generated with each dataset; instead of generating four policies, one could generate just one new policy. To achieve the same ensemble size, one would have to generate a new policy more often, i.e., after a smaller number of new observations. To have the same dataset size, the datasets would need to overlap. While this increases the correlation between policies generated subsequently, the correlation would not be as high as between four policies generated from the same dataset.

### 7.3 Summary

This chapter combined ideas from previous chapters to deal with changing environments. First, the case was considered where a collection of policies is already available, but it is unclear which of those policies are most suitable for the current environment. It was suggested to first run an ensemble with a random selection of policies to collect observations (while still maintaining a reasonable performance). The observations collected this way are then used for policy evaluation, resulting in estimates of a policy's performance on the current environment. The estimates are then used to select a subset of policies to form an ensemble subsequently controlling the environment. The experiments showed that ensembles of policies selected using policy assessment perform significantly better than randomly selected policies. An approach like this can prove useful whenever policies from one or more similar systems are already available and are to be transferred to another similar system. For example, if an operating company were to put a new turbine into operation, it could use policies from all other, already operated turbines as an initial pool of policies to select from.

In a second step the evolving ensemble approach was introduced. The evolving ensemble adapts to a changing environment by periodically adding new policies derived from recent observations and discarding policies that have become obsolete. The decision which policies to remove and which to keep can be based on a policy's age. Using policy assessment can provide a superior alternative. Both approaches were evaluated on the pole-balancing and cart-pole benchmark problems and compared with two simpler approaches, namely using a single policy learned from recent observations and an ensemble trained from the same observation set. The results indicated that the evolving ensemble is superior, especially in terms of minimum performance. The results from the pole-balancing experiments

showed that the evolving ensemble approach can considerably benefit from policy assessment—no other approach could deliver such a high and consistent performance. On the other hand, the experiments on the cart-pole problem revealed that uncorrelated or even negatively correlated policy assessment can weaken the ensemble. Also the remove-old evolving ensemble showed weaknesses in this setting—if two or more subsequent updates add poor policies, a notable performance drop might occur. The evolving ensemble using policy assessment did not show this problem, although the policy assessment was almost uncorrelated to the true policy performance, thus effectively removing policies randomly. To confirm this assumption, the same experiment was performed with actual random selection of policies. It turned out that this approach performed comparably and was able to avoid the performance drop of the remove-old evolving ensemble. The reason for this is the wider range that policies are used from, thus decreasing the correlation between policies in the ensemble. From this follows the advice to use large ensembles, preferably with members from different timeframes and therefore different observations.

An evolving ensemble is applicable whenever RL is applied to problems of autonomous control. Consider the example from above again, where a new gas turbine is to be put into operation. The existing policies from other turbines could serve as an initial policy pool. First, an ensemble of randomly selected policies could be used to collect observations. Those observations could be used to perform policy assessment of all policies and select the most suitable ones. Thereafter the ensemble could be updated by periodically adding new policies trained from recent observations and removing unsuitable policies according to policy assessment. No matter how fast the actual change of the system, due to the steady updates the evolving ensemble would always provide a suitable policy.

# 8

## Conclusion

This chapter concludes the thesis. It summarizes the previous chapters, highlights the main contributions, and lastly gives ideas for possible future research.

### 8.1 Summary

The thesis was concerned with reinforcement learning (RL) and its application to autonomous control. RL is an attractive method for autonomous control, because it learns from actual observations of the system and can thus deal with problems where classical approaches of optimal control are unfeasible. Furthermore, an RL method can adapt to changing characteristics of the system. Data-efficient RL methods have been developed in recent years, often employing powerful function approximators like neural networks. Although impressive results could be achieved, it was argued that the application of such methods is still cumbersome. In particular, often the learning process needs to be monitored and the policy as final result often has to be evaluated by executing it on an actual system. This hinders the application of RL to autonomously learning controllers that not only control the system in closed loop, but also autonomously update the control policy.

The thesis focused on increasing robustness, in particular decreasing the probability of obtaining poor and insufficient policies. Starting with discrete domains the issues of uncertainty were discussed. When estimating a Markov decision process (MDP) from a limited number of observations with the aim of obtaining an optimal policy for the underlying MDP, ignoring the uncertainties can have drastic consequences. The main reason for this is the maximization of the  $Q$ -function done in the Bellman optimality equation. This causes a positively biased  $Q$ -function, which leads to a policy that is, regarding the real, underlying MDP, often too optimistic. As a result, it might perform insufficiently on the real MDP. By incorporating the uncertainties of the MDP estimation, it becomes possible to obtain the uncertainties of the  $Q$ -function. This allows deriving policies that account for the estimates' uncertainties they are based on (quantile-optimal policies). To decrease the probability of deriving a policy that will perform poorly

on the real MDP, one optimizes a lower quantile than the expectation. The resulting policy realizes a trade-off between an action's return and the return's certainty. For example, from two actions with similar expected return the action with the more certain return would be chosen. A Monte Carlo approach was introduced that allows to determine the  $Q$ -function's uncertainties from the MDP estimate's uncertainties by considering a distribution over MDPs. As a more direct method to arrive at the  $Q$ -function's uncertainty, work by Schneegaß, Udfluft, and Martinetz (2008) (full-matrix uncertainty propagation (UP)) was described. They use uncertainty propagation to pass the transition and reward estimators' uncertainties to the result, i.e., the  $Q$ -function. Both, the Monte Carlo approach as well as full-matrix UP are computationally very expensive. As a feasible alternative, an approximate method, the diagonal approximation of uncertainty-incorporating policy iteration (DUIPI), was introduced. Uncertainty propagation uses a covariance matrix, whose size is the dominating factor in the computational requirements of full-matrix UP. By considering only the diagonal of the covariance matrix, DUIPI has the same complexity as the standard Bellman iteration (i.e., value iteration). Although being only approximate, it can improve upon the (uncertainty-ignorant) standard Bellman iteration and in particular decrease the risk of obtaining a very poor policy (quality assurance). In addition to the application of uncertainty awareness to quality assurance, we discussed the evaluation of a policy without actually executing it (self-assessment). Again, the consideration of uncertainty is important, since a  $Q$ -function derived from an MDP estimate is often positively biased. Our experiments showed that if the uncertainty is ignored, no reasoning about a policy's quality is possible using the  $Q$ -function alone. If, instead of evaluating only the  $Q$ -values, one considers the  $Q$ -values together with their uncertainties, i.e., evaluates a policy based on its  $Q$ -values minus the  $\varepsilon$ -weighted uncertainties, a more realistic policy evaluation becomes possible. As a third application, it was shown that uncertainty awareness can as well be used for efficient, uncertainty-seeking exploration. This can be achieved by changing the sign of  $\xi$ , the parameter weighting the uncertainty. This way the uncertainty acts as a bonus.

Since many interesting industrial control problems, including gas turbine control, feature a continuous state space, starting in Chapter 5 those problems were considered. The thesis focused on neural fitted  $Q$ -iteration (NFQ) as a neural RL method requiring only little manual monitoring during learning—automatically monitoring the error on some validation set and training the network in each iteration until the error is small is usually sufficient. However, the errors of the regression task in an NFQ iteration are by no means a suitable measure for policy quality. Thus it is hard to know when a good policy has been obtained, and therefore when to stop iterating. Simply iterating “long enough” is no viable option, since the policy quality often oscillates while iterating. This is one of the problems of NFQ; another is its tendency to overestimate the  $Q$ -function. Overall, while NFQ often delivers excellent policies in a data-efficient way, it cannot straightforwardly be used for autonomous control, if a considerable risk of

obtaining a poor policy cannot be accepted. As a possible solution the usage of ensembles was proposed. Ensembles are an established method in supervised learning. By combining different individual learners an ensemble is created that usually performs better than its members and often even better than any of the individual learners. Ensembles are effective if the members are accurate and diverse, i.e., make reasonable, although not excellent predictions and do not make the same errors for a given input. Surprisingly, so far the interest in ensembles in RL has been quite low; in particular, to the best of our knowledge a combination of NFQ with ensembles has not been considered before. Multiple ways of using ensembles in an NFQ context were discussed and evaluated. In the according experiments simple majority voting proved to be particularly effective. To realize a majority voting ensemble of NFQ policies, NFQ is run multiple times using the same dataset, each run generating a member of the final policy ensemble. The final policy ensemble is run by querying each individual policy for an action and selecting the action that most ensemble members chose. The variation introduced by the randomness in the training process of neural networks (random initial weights, stochastic pattern selection during training) as well as randomly splitting the data into training and validation set introduced a sufficient amount of diversity for ensembles to be effective. In one experiment, additionally a different network topology was used. It showed that with heterogeneous ensembles, consisting of members using different topologies, further improvements are possible. Still, homogeneous ensembles have the advantage of being simpler: once a regular NFQ implementation is available, it merely has to be run multiple times to produce the ensemble members. Since they deliver significant improvements over single network policies as well, in practice homogeneous ensembles are an appropriate choice. Ensembles work, because they suppress poor policies, the errors of individual members are “averaged out” similarly to classic ensembles for supervised learning problems. An ensemble can be expected to be always at least as good as a randomly selected member. If the quality of the individual policies varies, ensembles will likely perform better. In the experiments, ensembles lead to considerable improvements in all of the domains. In the scope of the thesis, only methods dealing with discrete actions could be considered. As a starting point for further research, two methods for continuous action selection in ensembles were sketched.

The hope was to be able to adapt the methods from the discrete setting to the continuous one to derive quantile-optimal policies there as well, i.e., explicitly consider the uncertainty and prefer actions with certain returns over uncertain ones. Unfortunately, adapting the methods to the continuous setting was not possible (at least not in a straightforward way), since one deals with different uncertainties in the different settings. In the discrete setting, one estimates an MDP (a model of the environment) from observations. Here, only the MDP estimate is affected by uncertainty; anything that follows, e.g., determining an optimal policy, is completely deterministic and not affected by uncertainty at all. One can access the MDP estimate’s uncertainties and use them to derive

the  $Q$ -function's uncertainties and thus determine quantile-optimal policies. In the continuous setting, one does not explicitly estimate a model. Instead, the  $Q$ -function is learned directly. In this case, the result is affected by two types of uncertainty: First, we have the uncertainty about the environment. This is similar to the discrete setting, but this time one cannot access it directly, since no explicit model of the environment is used. Second, uncertainty is introduced by the learning process itself (due to stochastic influences like random start weights and random pattern selection). By using the Monte Carlo approach of repeatedly running the learning process one can estimate this uncertainty. Unfortunately, it cannot be used to derive quantile-optimal policies, since for this one needs to know the uncertainty contained within the observations as well. Nonetheless, ensembles achieve something similar as quantile-optimal policies for quality assurance: by suppressing poor member policies they decrease the probability of obtaining a poor overall policy. Analogously, a quantile-optimal policy in the discrete setting decreases the probability of obtaining a poor policy by restricting the use of uncertain knowledge about an MDP.

In Chapter 6 methods for self-assessment in continuous domains were discussed, i.e., evaluating a policy without actually executing it. The  $Q$ -function generated by NFQ is not a good indicator for policy quality, which could be seen by comparing the  $Q$ -function and the policy's true performance. Because of the inability of obtaining the  $Q$ -function's uncertainty, it was necessary to look for other ways of arriving at a more meaningful quality measure than the  $Q$ -function. Experiments with fitted policy evaluation (FPE) were performed, an adaptation of the fitted  $Q$ -iteration (FQI) approach to policy evaluation. It turned out that using neural fitted policy evaluation (NFPE) with the same set of observations already leads to a better performance measure. By using a different dataset and/or a different function approximator one can further improve the results. In practice, especially the approach using a different function approximator is attractive, since in this case no additional data for evaluation is required. It was shown that a performance measure derived this way is more informative than the  $Q$ -function from NFQ. In experiments the performance measure was used to select good or reject poor policies from a set of given policies. It was further used to weight ensemble members, leading to a further increase of the performance of ensemble policies.

In Chapter 7 the ideas from previous chapters were combined to deal with changing environments. First the situation was considered where one already has a set of policies trained for different versions of a certain environment. It was assumed that it was unclear which policies from the set are most suitable for a given environment. In such a setting, one can first select policies randomly and use them as an initial ensemble, which is run for a number of steps. The resulting observations are then used to evaluate all available policies and then select the supposedly best ones. This method was evaluated using the pole-balancing benchmark, where different pole masses were used as different versions of the environment. It showed that ensembles of policies selected by FPE performed much better than ensembles of randomly selected policies. Finally, the idea of

the evolving ensemble was introduced. It adapts to slowly changing environment by continuously adding new policies that are trained using recent observations. At the same time policies that have become unsuitable are removed from the ensemble. Two methods for removing policies were considered: simply removing the oldest policies and FPE-based removal of policies that have become obsolete. The approaches were evaluated on the pole-balancing and cart-pole benchmarks and were compared with less sophisticated approaches, namely single policies and simple (non-evolving) ensembles. On the pole-balancing benchmark the evolving approaches were clearly superior, in particular when looking at the minimum over all trials. On the cart-pole benchmark the situation was not as clear. While the evolving approaches here as well delivered a better and more consistent performance, the approach using FPE could not show advantages. Further analysis showed that in this setting tree-based FPE was unable to generate useful performance measurements, resulting in random selection of policies. Still, this approach was able to avoid a severe performance drop that occurred when for a number of successive iterations poor policies were generated. This problem can be circumvented by using proper FPE to select policies. If this is not available, as was the case in the cart-pole experiment, one should include policies from a time range as wide as possible in the ensemble to maximize its diversity. The experiments with the pole-balancing benchmark showed that a means of policy evaluation can be a great addition. When applying the evolving ensemble approach, one should first try to find such a means. Even if the correlation is only small, it will improve the evolving ensemble.

## 8.2 Contributions

In the following, the major contributions of this thesis are outlined:

- (i) **Monte Carlo estimate of a  $Q$ -function distribution.** As a first and direct way to estimate a  $Q$ -function's uncertainty, a Monte Carlo approach was presented. The approach estimates not only the uncertainty, but a complete  $Q$ -function distribution. It further illustrates how the uncertainty of the MDP estimate influences the  $Q$ -function's uncertainty. It was further argued that the Monte Carlo approach estimates the true distribution. The other approaches, i.e., using uncertainty propagation to determine the  $Q$ -function's uncertainties from those of the MDP estimate (full-matrix UP) and its fast approximation using only the diagonal of the covariance matrix (DUIPI), can only approximate the true uncertainty because of  $\mathbf{E}_i f(x_i) \neq f(\mathbf{E}_i x_i)$ , if  $f$  is a non-linear function (in our case the repeated application of the Bellman operator). While the left-hand side of the equation is what we are looking for, the direct methods can only deliver the right hand side. The Monte Carlo estimate served mainly to illustrate that the  $Q$ -function's uncertainty stems from the uncertainty about the MDP, if the true MDP

is unknown and only observations are available. Although of the presented methods the Monte Carlo estimate is the only one that delivers the true distribution, in practice one of the other methods will be preferable due to the Monte Carlo estimate's high computational requirements.

- (ii) **Diagonal approximation of uncertainty incorporating policy iteration (DUIPI).** With DUIPI, a method to efficiently approximate the  $Q$ -function's uncertainty was presented. By neglecting the non-diagonal elements of the covariance matrix it has the same complexity as the standard Bellman iteration. A number of experiments were conducted that showed the effectiveness of the method for quality assurance, i.e., reducing the probability of obtaining a poor policy.

DUIPI should be considered whenever one deals with an MDP with discrete state and action spaces and is looking for an uncertainty-aware method. The knowledge of uncertainty can then be used to lower the probability of obtaining a poor policy because the wrong MDP has been estimated. Although DUIPI is approximate (due to its ignorance of non-diagonal elements of the covariance matrix), it is the preferred method for domains with more than 100 states, since the computational requirements limit the applicability of full-matrix uncertainty propagation.

- (iii) **Using uncertainty awareness for exploration and self-assessment.** It was shown that in addition to quality assurance the concept of uncertainty awareness can be used for efficient, uncertainty-seeking exploration. In this context the diagonal approximation of uncertainty-incorporating policy iteration with  $Q$  modification (DUIPI-QM) was introduced, a method that modifies the  $Q$ -values itself to include uncertainty. The methods were evaluated experimentally on a number of benchmark problems from the literature and they were compared with established exploration techniques. It turned out that they perform comparably, while having the advantage of using the notion of uncertainty explicitly and allowing to use the same method for both, exploration and quality assurance. Additionally, it was discussed how the  $Q$ -function can be used to assess a policy's quality without actually executing it. It was shown that the  $Q$ -function itself is often positively biased and that considering the uncertainty mitigates this problem as well.

Uncertainty-based exploration is an alternative to established discrete-domain exploration methods like R-Max or model-based interval estimation (MBIE). Compared to those it has the advantage that it is based on the same method that can be used for quality assurance; the only difference is the setting of the parameter weighting the uncertainty—if it is set to a positive value, the resulting policy tries to avoid uncertainty (quality assurance); set to a negative value, an uncertainty-seeking policy results (exploration).

- (iv) **Combination of NFQ and ensembles.** With the combination of NFQ and ensembles, a novel approach was proposed that makes NFQ more reli-

able and thus more suitable for autonomous control. Several ways of using ensembles in an NFQ context were discussed and evaluated; it turned out that majority voting is a simple yet effective approach.

Introducing ensembles to NFQ is maybe the most important contribution towards autonomous RL. NFQ by itself is a very powerful approach to tackle RL problems with continuous (and potentially high-dimensional) state spaces in a data-efficient way. However, the reliability issue of NFQ remains; using ensembles allows to significantly weaken, if not overcome this issue.

- (v) **Self-assessment in continuous domains.** When evaluating methods for self-assessment in continuous domains, it showed that the  $Q$ -function generated by NFQ is a poor indicator for policy quality. It was proposed to use FPE for a given policy, and results were presented indicating that using FPE the correlation between true performance and the quality measure is increased. Moreover, it was proposed to use a different function approximator than used to generate the policy, since a different method is less likely to make the same errors. This idea can be combined with using a different set of observations as well.

Evaluating the quality of a policy is important when applying RL to autonomous control, since employing an insufficient policy can lead to unacceptable performance loss or even damage. To the best of the author's knowledge, in the context of NFQ so far the quality of the  $Q$ -function as a performance indicator has not been considered. As it showed that the NFQ  $Q$ -function alone is insufficient to derive a performance measure, the proposed alternatives, i.e., using a different function approximator and/or a different dataset for policy evaluation, are important ingredients to allow the application of RL to autonomous control.

- (vi) **The evolving ensemble as an approach to autonomous control in changing environments.** Finally, the evolving ensemble approach was presented. By combining ensembles and self-assessment it is suitable for autonomous control in changing environments, because it continuously adapts to the environment by adding new policies and dropping those that have become obsolete. Also without a means of policy assessment the evolving ensemble can be used, in this case by dropping the oldest policies.

The evolving ensemble advances the state of the art of applying RL to environments whose characteristics slowly change. While the traditional approach of periodically learning a new policy from recent data can lead to abruptly changing policies and, at least in the case of NFQ, occasionally produce a poor policy, the policy of the evolving ensemble changes gradually, since from one ensemble generation to the next only a subset of policies is replaced. This allows to maintain an ensemble policy of high quality,

especially when using policy assessment to decide which policies to keep and which to replace.

### 8.3 Future Research

Finally, ideas for future research are discussed.

The discrete domain methods were mainly evaluated empirically through experiments. Theoretical results would be desirable as well, in particular to quantify the loss introduced by DUIPI’s ignorance of the non-diagonal elements of the covariance matrix in comparison to the full-matrix algorithm. Further it would be interesting to see what exactly the effects of calculating  $f(\mathbf{E}_i x_i)$  instead of  $\mathbf{E}_i f(x_i)$  are, i.e., applying the Bellman iteration to the expected MDP instead of taking the expectation of the Bellman iteration applied to all possible MDPs according to their distribution. The Monte Carlo approach approximates the latter and reveals the true distribution given an unlimited number of samples.

Since the thesis’s major background was optimal control of technical systems like gas turbines, instead of further dealing with methods for discrete state spaces and addressing open questions, the thesis moved on to continuous state methods, as the regarded real-world problems also feature a continuous state space.

In the context of ensembles for RL, directions of future research include the combination of different methods. In this work, only NFQ was used. It would be interesting to see how ensembles with policies from different methods perform, for example variants of rewards regression (Schneegaß, Udluft, and Martinetz, 2007b; Schneegaß, Udluft, and Martinetz, 2007a). Moreover, one could use FQI with different function approximators than neural networks, for example combining policies from extra-tree FQI (Ernst, Geurts, and Wehenkel, 2005) and NFQ in an ensemble. Heterogeneous ensembles that use different neural topologies were briefly experimented with. This idea could be taken one step further by using random topologies. Randomizing the members’ topologies could potentially increase the diversity of the ensemble and thus lead to even better results. The same holds for learning parameters like the learning rate or the learning algorithm for training a neural network—those could be chosen at random as well. However, care must be taken to still obtain *accurate* ensemble members. Furthermore, experiments with other domains should be performed to get a better understanding of where ensemble method are most effective. Also, a thorough theoretical analysis could lead to useful insights. Moreover, the influence of the exploration method could be investigated. In the experiments random exploration was used to obtain a fairly equal distribution of observations over the state space. How do ensemble methods behave if the distribution is skewed? Could re-sampling help in such a situation? Finally, only discrete action problems were considered and ideas for ensembles in continuous action problems were only sketched. It would be worthwhile to look into those problems as well, since many real-world

problems feature continuous action spaces.

The evolving ensemble approach as well offers several directions for further research. First, other domains and settings could be evaluated. Moreover, more realistic exploration schemes should be evaluated—to focus on the approach, random exploration was included to avoid the problem of imbalanced data. One could try to find criteria for when imbalanced data are problematic and evaluate methods to avoid the imbalance (like re-sampling). Further experiments could also lead to insights in the influence of ensemble size, the time-frame policies are chosen from, and other parameters.

This thesis’s aim was to advance the applicability of RL to autonomously learning controllers in potentially changing environments. The methods worked well for the considered benchmark domains. A major direction for future research will be applying them to an actual system.

Research in artificial intelligence has at least in part always been concerned with methods that are able to adapt and learn, because being able to learn must be considered part of intelligent behavior. RL in particular is a method that closely resembles the way humans learn. Sutton and Barto write:

“When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment. Exercising this connection produces a wealth of information about cause and effect, about the consequences of actions, and about what to do in order to achieve goals.” (Sutton and Barto, 1998, p. 3)

When applying RL to complex control tasks, we formulate our goals through the reward function. If all goes well, the agent then *magically* learns what to do to achieve those goals. Through its contributions this thesis tried to increase the odds that this *magic* actually happens.



# Bibliography

- Abbeel, P., A. Coates, M. Quigley, and A. Y. Ng (2006). “An Application of Reinforcement Learning to Aerobatic Helicopter Flight”. In: *Proceedings of the 20th Conference on Neural Information Processing Systems*. MIT Press, pp. 1–8 (cit. on p. 2).
- Abtahi, F. and I. Fasel (2011). “Deep Belief Nets as Function Approximators for Reinforcement Learning”. In: *Lifelong Learning: Papers from the 2011 AAAI Workshop (WS-11-15)*, pp. 2–7 (cit. on p. 113).
- Albus, J. (1971). “A theory of cerebellar function”. In: *Mathematical Biosciences* 10.1-2, pp. 25–61 (cit. on p. 26).
- Atkeson, C. and J. Santamaria (1997). “A comparison of direct and model-based reinforcement learning”. In: *Proceedings of the 1997 IEEE International Conference on Robotics and Automation*. Vol. 4. IEEE, pp. 3557–3564 (cit. on p. 23).
- Barto, A., R. Sutton, and C. Anderson (1983). “Neuronlike adaptive elements that can solve difficult learning control problems”. In: *IEEE Transactions on systems, man, and cybernetics* 13.5, pp. 834–846 (cit. on p. 82).
- Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press (cit. on pp. 1, 5, 18).
- Bertsekas, D. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall (cit. on p. 20).
- Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press (cit. on pp. 25, 76).
- Borkar, S. et al. (2005). *Platform 2015: Intel processor and platform evolution for the next decade*. Tech. rep. Intel White Paper (cit. on p. 94).
- Brafman, R. and M. Tenenholz (2003). “R-max - a general polynomial time algorithm for near-optimal reinforcement learning”. In: *Journal of Machine Learning Research* 3, pp. 213–231 (cit. on pp. 29, 63–65).
- Breiman, L. (1996). “Bagging predictors”. In: *Machine Learning* 24.2, pp. 123–140 (cit. on pp. 78, 79).
- Brown, G., J. Wyatt, R. Harris, and X. Yao (2005). “Diversity creation methods: a survey and categorisation”. In: *Information Fusion* 6.1, pp. 5–20 (cit. on p. 79).
- Buşoniu, L., R. Babuška, B. D. Schutter, and D. Ernst (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximation*. Boca Raton, FL: CRC Press (cit. on pp. 1, 30).

- Congdon, P. (2006). *Bayesian statistical modelling*. Wiley series in probability and mathematical statistics. Probability and mathematical statistics. John Wiley & Sons (cit. on p. 35).
- Coppersmith, D. and S. Winograd (1990). “Matrix multiplication via arithmetic progressions”. In: *Journal of Symbolic Computation* 9, pp. 251–280 (cit. on p. 41).
- Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2009). *Introduction to Algorithms, Third Edition*. MIT Press (cit. on p. 18).
- D’Agostini, G. (2003). *Bayesian Reasoning in Data Analysis: A Critical Introduction*. World Scientific Publishing (cit. on pp. 5, 40).
- Dearden, R., N. Friedman, and D. Andre (1999). “Model based Bayesian Exploration”. In: *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pp. 150–159 (cit. on pp. 35, 56, 60, 64).
- Dearden, R., N. Friedman, and S. J. Russell (1998). “Bayesian Q-Learning”. In: *Proceedings of AAAI/IAAI*, pp. 761–768 (cit. on pp. 63, 64).
- Dietterich, T. G. (2000). “Ensemble methods in machine learning”. In: *Multiple classifier systems*, pp. 1–15 (cit. on pp. 6, 76, 78, 90, 100).
- Dijkstra, E. W. (1959). “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1 (1), pp. 269–271 (cit. on p. 18).
- Duell, S., A. Hans, and S. Udluft (2010). “The Markov Decision Process Extraction Network”. In: *Proceedings of the 18th European Symposium on Artificial Neural Networks* (cit. on pp. 11, 15).
- Duell, S., L. Weichbrodt, A. Hans, and S. Udluft (2012). “Recurrent Neural State Estimation in Domains with Long-Term Dependencies”. In: *Proceedings of the 20th European Symposium on Artificial Neural Networks* (cit. on p. 11).
- Efron, B. (1979). “Bootstrap methods: another look at the jackknife”. In: *The Annals of Statistics* 7.1, pp. 1–26 (cit. on p. 79).
- Efron, B. and R. Tibshirani (1993). *An introduction to the bootstrap*. Vol. 57. Chapman & Hall/CRC (cit. on p. 79).
- Eiben, A. E. and J. E. Smith (2008). *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer (cit. on p. 28).
- Ernst, D., P. Geurts, and L. Wehenkel (2003). “Iteratively extending time horizon reinforcement learning”. In: *Machine Learning: ECML 2003, 14th European Conference on Machine Learning*, pp. 96–107 (cit. on pp. 26, 71).
- Ernst, D., P. Geurts, and L. Wehenkel (2005). “Tree-Based Batch Mode Reinforcement Learning”. In: *Journal of Machine Learning Research* 6, pp. 503–556 (cit. on pp. 6, 26, 71, 80, 81, 132).

- Faußer, S. and F. Schwenker (2011). “Ensemble Methods for Reinforcement Learning with Function Approximation”. In: *Multiple Classifier Systems: 10th International Workshop*. Springer, pp. 56–65 (cit. on p. 80).
- Freund, Y., R. Schapire, and N. Abe (1999). “A short introduction to boosting”. In: *Journal of Japanese Society for Artificial Intelligence* 14, pp. 771–780 (cit. on pp. 78, 79).
- Friedman, J. (2001). “Greedy function approximation: a gradient boosting machine”. In: *The Annals of Statistics* 29.5, pp. 1189–1232 (cit. on p. 79).
- Friedman, N. and Y. Singer (1999). “Efficient Bayesian Parameter Estimation in Large Discrete Domains”. In: *Advances in Neural Information Processing Systems*. MIT Press (cit. on p. 34).
- Gabel, T., C. Lutz, and M. Riedmiller (2011). “Improved Neural Fitted Q Iteration Applied to a Novel Computer Gaming and Learning Benchmark”. In: *Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE Press (cit. on p. 75).
- Gabel, T. and M. Riedmiller (2006). “Reducing policy degradation in neurodynamic programming”. In: *Proceedings of the European Symposium on Artificial Neural Networks*, pp. 653–658 (cit. on pp. 6, 61, 73).
- Gaskett, C. (2002). “Q-learning for robot control”. PhD thesis. The Australian National University (cit. on pp. 6, 73, 74).
- Geibel, P. (2001). “Reinforcement Learning with Bounded Risk”. In: *Proceedings of the 18th Int. Conf. on Machine Learning*. Morgan Kaufmann, San Francisco, CA, pp. 162–169 (cit. on p. 65).
- Geman, S., E. Bienenstock, and R. Doursat (1992). “Neural networks and the bias/variance dilemma”. In: *Neural Computation* 4(1), pp. 1–58 (cit. on p. 79).
- Geurts, P., D. Ernst, and L. Wehenkel (2006). “Extremely Randomized Trees”. In: *Machine Learning* 63.1, pp. 3–42 (cit. on pp. 71, 100).
- Gomez, F. and R. Miikkulainen (1999). “Solving non-Markovian control tasks with neuroevolution”. In: *International Joint Conference on Artificial Intelligence*. Vol. 16, pp. 1356–1361 (cit. on p. 28).
- Gordon, G. J. (1995). “Stable Function Approximation in Dynamic Programming”. In: *Proceedings of the International Conference on Machine Learning* (cit. on p. 26).
- Gordon, G. J. (2001). “Reinforcement learning with function approximation converges to a region”. In: *Advances in Neural Information Processing Systems*, pp. 1040–1046 (cit. on pp. 6, 73).

- Hafner, R. and M. Riedmiller (2011). “Reinforcement learning in feedback control”. In: *Machine Learning*, pp. 1–33 (cit. on p. 27).
- Hairer, E., S. Norsett, and G. Wanner (2002). *Solving Ordinary Differential Equations I*. Springer (cit. on p. 83).
- Hans, A., S. Duell, and S. Udluft (2011). “Agent Self-Assessment: Determining Policy Quality Without Execution”. In: *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning* (cit. on p. 10).
- Hans, A., D. Schneegaß, A. M. Schäfer, and S. Udluft (2008). “Safe Exploration for Reinforcement Learning”. In: *Proceedings of the 16th European Symposium on Artificial Neural Networks*, pp. 413–418 (cit. on p. 11).
- Hans, A. and S. Udluft (2009). “Efficient Uncertainty Propagation for Reinforcement Learning with Limited Data”. In: *Proceedings of the International Conference on Artificial Neural Networks* (cit. on p. 10).
- Hans, A. and S. Udluft (2010a). “Ensembles of Neural Networks for Robust Reinforcement Learning”. In: *Proceedings of the 9th IEEE International Conference on Machine Learning and Applications*. IEEE, pp. 401–406 (cit. on p. 10).
- Hans, A. and S. Udluft (2010b). “Uncertainty Propagation for Efficient Exploration in Reinforcement Learning”. In: *Proceedings of the 19th European Conference on Artificial Intelligence*. IOS Press (cit. on p. 10).
- Hans, A. and S. Udluft (2011). “Ensemble Usage for More Reliable Policy Identification in Reinforcement Learning”. In: *Proceedings of the 19th European Symposium on Artificial Neural Networks* (cit. on p. 11).
- Hansen, L. and P. Salamon (1990). “Neural Network Ensembles”. In: *IEEE Transactions Pattern Analysis and Machine Intelligence* 12.10, pp. 993–1001 (cit. on p. 78).
- Hastie, T., R. Tibshirani, and J. Friedman (2001). *The Elements Of Statistical Learning Theory: Data Mining, Inference, and Prediction*. New York: Springer (cit. on pp. 1, 6, 71).
- Heger, M. (1994). “Consideration of risk in reinforcement learning”. In: *Proceedings of the 11th Int. Conf. on Machine Learning*. Morgan Kaufmann, pp. 105–111 (cit. on p. 65).
- Jacobs, R., M. Jordan, S. Nowlan, and G. Hinton (1991). “Adaptive mixtures of local experts”. In: *Neural computation* 3.1, pp. 79–87 (cit. on p. 78).
- Jiang, J. and M. Kamel (2006). “Aggregation of reinforcement learning algorithms”. In: *Neural Networks, 2006. IJCNN'06. International Joint Conference on*. IEEE, pp. 68–72 (cit. on p. 80).

- Kaelbling, L., M. Littman, and A. Cassandra (1998). “Planning and acting in partially observable stochastic domains”. In: *Artificial Intelligence* 101.1-2, pp. 99–134 (cit. on p. 15).
- Kaelbling, L., M. Littman, and A. Moore (1996). “Reinforcement Learning: A Survey”. In: *Journal of Artificial Intelligence Research* 4, pp. 237–285 (cit. on pp. 15, 30).
- Kalyanakrishnan, S. and P. Stone (2007). “Batch Reinforcement Learning in a Complex Domain”. In: *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems* (cit. on p. 54).
- Kearns, M. and S. Singh (1998). “Near-Optimal Reinforcement Learning in Polynomial Time”. In: *Proceedings of the International Conference on Machine Learning*, pp. 260–268 (cit. on pp. 29, 64).
- Kober, J. and J. Peters (2012). “Reinforcement Learning in Robotics: A Survey”. In: *Reinforcement Learning: State of the Art*. Ed. by M. Wiering and M. Otterlo. Vol. 12. Adaptation, Learning, and Optimization. Springer, pp. 579–610 (cit. on p. 2).
- Krogh, A. and J. Vedelsby (1995). “Neural Network Ensembles, Cross Validation, and Active Learning”. In: *Advances in Neural Information Processing Systems*. Vol. 7, pp. 231–238 (cit. on pp. 78, 79, 100).
- Kruger, J. and D. Dunning (1999). “Unskilled and unaware of it: How difficulties in recognizing one’s own incompetence lead to inflated self-assessments.” In: *Journal of personality and social psychology* 77.6, p. 1121 (cit. on p. 99).
- Lagoudakis, M. G. and R. Parr (2003). “Least-Squares Policy Iteration”. In: *Journal of Machine Learning Research*, pp. 1107–1149 (cit. on p. 26).
- Lala, J. and R. Harper (1994). “Architectural principles for safety-critical real-time applications”. In: *Proceedings of the IEEE* 82.1, pp. 25–40 (cit. on p. 100).
- Lee, H. et al. (2006). “Quadruped Robot Obstacle Negotiation via Reinforcement Learning”. In: *Proceedings of the 2006 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3003–3010 (cit. on p. 2).
- Levenshtein, V. (1966). “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet Physics Doklady* 10 (cit. on p. 18).
- Lin, L. (1992). “Self-improving reactive agents based on reinforcement learning, planning and teaching”. In: *Machine Learning* 8.3, pp. 293–321 (cit. on p. 23).
- Merke, A. and M. Riedmiller (2001). “Karlsruhe Brainstormers - A Reinforcement Learning Approach to Robotic Soccer”. In: *RoboCup 2001: Robot Soccer World Cup V*. Springer, pp. 435–440 (cit. on p. 2).

- Metropolis, N. and S. Ulam (1949). “The Monte Carlo Method”. In: *Journal of the American Statistical Association* 44, pp. 335–341 (cit. on p. 35).
- Michie, D. and R. Chambers (1968). “BOXES: An experiment in adaptive control”. In: *Machine intelligence 2.2*, pp. 137–152 (cit. on p. 82).
- Migliavacca, M. et al. (2010). “Fitted Policy Search: Direct Policy Search Using a Batch Reinforcement Learning Approach”. In: *Proceedings of the 3rd International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems* (cit. on pp. 62, 98).
- Migliavacca, M. et al. (2011). “Fitted Policy Search”. In: *Proceedings of the Symposium on Adaptive Dynamic Programming And Reinforcement Learning*. IEEE (cit. on pp. 28, 62, 98).
- Moore, A. W. and C. G. Atkeson (1993). “Prioritized Sweeping: Reinforcement Learning With Less Data and Less Time”. In: *Machine Learning* 13, pp. 103–130 (cit. on p. 24).
- Neuneier, R. and O. Mihatsch (1998). “Risk Sensitive Reinforcement Learning”. In: *Advances in Neural Information Processing Systems*, pp. 1031–1037 (cit. on p. 65).
- Neuneier, R. and H. G. Zimmermann (1998). “How to Train Neural Networks”. In: *Neural Networks: Tricks of the Trade*. Ed. by G. B. Orr and K.-R. Mueller. Berlin: Springer Verlag, pp. 373–423 (cit. on p. 75).
- Ormoneit, D. and S. Sen (2002). “Kernel-based reinforcement learning”. In: *Machine Learning* 49.2-3, pp. 161–178 (cit. on p. 26).
- Peng, J. and R. Williams (1996). “Incremental multi-step Q-learning”. In: *Machine Learning* 22.1, pp. 283–290 (cit. on p. 24).
- Peters, J. and S. Schaal (2008). “Reinforcement learning of motor skills with policy gradients”. In: *Neural Networks* 21.4, pp. 682–697 (cit. on p. 2).
- Puterman, M. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons Canada, Ltd. (cit. on p. 2).
- Rasmussen, C. E. and C. K. I. Williams (2006). *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press (cit. on p. 97).
- Riedmiller, M. (2005). “Neural Fitted Q-Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method”. In: *Proceedings of the 16th European Conf. on Machine Learning*, pp. 317–328 (cit. on pp. 3, 6, 26, 71, 72, 84, 86).
- Riedmiller, M., J. Peters, and S. Schaal (2007). “Evaluation of policy gradient methods and variants on the cart-pole benchmark”. In: *International Sym-*

- posium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE, pp. 254–261 (cit. on p. 27).
- Riedmiller, M. (2010). Reinforcement Learning Mailing List, <http://groups.google.com/group/rl-list/msg/f75e959729544f58> (cit. on p. 86).
- Riedmiller, M. (2011). Personal Communication (cit. on p. 74).
- Rummery, G. and M. Niranjan (1994). *On-line Q-learning using connectionist systems*. Tech. rep. Cambridge University (cit. on pp. 22, 24).
- Sato, M. and S. Kobayashi (2000). “Variance-Penalized Reinforcement Learning for Risk-Averse Asset Allocation”. In: *Proceedings of the International Conference on Intelligent Data Engineering and Automated Learning, Data Mining, Financial Engineering, and Intelligent Agents*. London, UK: Springer-Verlag, pp. 244–249 (cit. on p. 65).
- Schäfer, A. M. (2008). “Reinforcement Learning with Recurrent Neural Networks”. PhD thesis. Osnabrück University (cit. on p. 28).
- Schäfer, A. M. and S. Udluft (2005). “Solving Partially Observable Reinforcement Learning Problems With Recurrent Neural Networks”. In: *Workshop Proceedings of the European Conference on Machine Learning (ECML-05)*, pp. 71–81 (cit. on p. 15).
- Schneegaß, D., S. Udluft, and T. Martinetz (2006). “Kernel Rewards Regression: An Information Efficient Batch Policy Iteration Approach.” In: *Artificial Intelligence and Applications*. Ed. by V. Devedzic. IASTED/ACTA Press, pp. 428–433 (cit. on p. 3).
- Schneegaß, D., S. Udluft, and T. Martinetz (2008). “Uncertainty Propagation for Quality Assurance in Reinforcement Learning”. In: *Proceedings of the International Joint Conference on Neural Networks*, pp. 2589–2596 (cit. on pp. 5, 32, 44, 49, 51, 53, 58, 126).
- Schneegaß, D., S. Udluft, and T. Martinetz (2007a). “Neural rewards regression for near-optimal policy identification in Markovian and partial observable environments”. In: *Proceedings of the European Symposium on Artificial Neural Networks*, pp. 301–306 (cit. on pp. 3, 26, 132).
- Schneegaß, D., S. Udluft, and T. Martinetz (2006). “Kernel Rewards Regression: An Information Efficient Batch Policy Iteration Approach”. In: *Proceedings of the IASTED Conference on Artificial Intelligence and Applications*, pp. 428–433 (cit. on p. 26).
- Schneegaß, D., S. Udluft, and T. Martinetz (2007b). “Explicit Kernel Rewards Regression for Data-Efficient Near-optimal Policy Identification”. In: *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*. Ed. by M. Verleysen, pp. 337–342 (cit. on pp. 26, 132).

- Schneegaß, D., S. Udluft, and T. Martinetz (2007c). “Improving Optimality of Neural Rewards Regression for Data-efficient Batch Near-Optimal Policy Identification”. In: *Proceedings of the International Conference on Artificial Neural Networks* (cit. on p. 27).
- Schäfer, A., S. Udluft, and H.-G. Zimmermann (2007). “A Recurrent Control Neural Network for Data Efficient Reinforcement Learning”. In: *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. Honolulu, HI (cit. on p. 27).
- Schäfer, A. M., D. Schneegaß, V. Sterzing, and S. Udluft (2007). “A neural reinforcement learning approach to gas turbine control”. In: *Proceedings of the International Joint Conference on Neural Networks* (cit. on p. 2).
- Sehnke, F. et al. (2010). “Parameter-exploring policy gradients”. In: *Neural Networks* 23.4, pp. 551–559 (cit. on p. 27).
- Shannon, C. (1950). “Programming a Computer for Playing Chess”. In: *Philosophical Magazine* 41.314 (cit. on p. 2).
- Singh, S. (1993). “The efficient learning of multiple task sequences”. In: *Advances in Neural Information Processing Systems*, pp. 251–251 (cit. on pp. 80, 81).
- Stephan, V. et al. (2000). “A Reinforcement Learning Based Neural Multi-Agent-System for Control of a Combustion Process”. In: *Proceedings of the International Joint Conference on Neural Networks*, pp. 217–222 (cit. on p. 2).
- Strehl, A. and M. Littman (2009). “An analysis of model-based Interval Estimation for Markov Decision Processes.” In: *Journal of Computer and System Sciences* 74.8, pp. 1309–1331 (cit. on pp. 29, 30, 63–65).
- Sun, R. and T. Peterson (1999). “Multi-agent reinforcement learning: weighting and partitioning”. In: *Neural Networks* 12.4-5, pp. 727–753 (cit. on pp. 80, 81).
- Sutton, R. (1988). “Learning to predict by the methods of temporal differences”. In: *Machine Learning* 3.1, pp. 9–44 (cit. on pp. 22, 25, 26).
- Sutton, R. (1996). “Generalization in reinforcement learning: Successful examples using sparse coarse coding”. In: *Advances in Neural Information Processing Systems*, pp. 1038–1044 (cit. on p. 26).
- Sutton, R. and A. Barto (1998). *Reinforcement Learning: An Introduction*. MIT Press (cit. on pp. 1, 15, 20, 30, 133).
- Szepesvári, C. (2010). *Algorithms for Reinforcement Learning*. Morgan & Claypool Publishers (cit. on p. 30).
- Tesauro, G. J. (1994). “TD-gammon, a self-teaching backgammon program, achieves master-level play”. In: *Neural Computation* 6(2), pp. 215–219 (cit. on pp. 2, 25).

- Tham, C. (1995). “Reinforcement learning of multiple tasks using a hierarchical CMAC architecture”. In: *Robotics and Autonomous Systems* 15.4, pp. 247–274 (cit. on pp. 80, 81).
- Thrun, S. and A. Schwartz (1993). “Issues in using function approximation for reinforcement learning”. In: *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum* (cit. on pp. 6, 73, 74).
- Thrun, S. (1992). *Efficient exploration in reinforcement learning*. Tech. rep. School of Computer Science, Carnegie Mellon University (cit. on p. 29).
- Tresp, V. (1994). *The Wet Game of Chicken*. Tech. rep. Siemens AG, CT IC 4 (cit. on pp. 51, 56).
- Tromp, J. (2010). *John’s Chess Playground*. <http://www.cwi.nl/~tromp/chess/chess.html> (cit. on p. 2).
- Tsitsiklis, J. (1994). “Asynchronous stochastic approximation and Q-learning”. In: *Machine Learning* 16.3, pp. 185–202 (cit. on p. 26).
- Van Hasselt, H. (2010a). Personal Communication (cit. on p. 81).
- Van Hasselt, H. (2010b). “Double Q-learning”. In: *Advances in Neural Information Processing Systems* (cit. on p. 74).
- Watkins, C. (1989). “Learning from Delayed Rewards”. PhD thesis. University of Cambridge (cit. on pp. 2, 23, 24, 26).
- Whiteson, S. (2012). “Evolutionary Computation for Reinforcement Learning”. In: *Reinforcement Learning: State of the Art*. Ed. by M. Wiering and M. Otterlo. Vol. 12. Adaptation, Learning, and Optimization. Springer, pp. 579–610 (cit. on p. 28).
- Widrow, B. and M. Hoff (1960). “Adaptive switching circuits”. In: *1960 WESCON Convention Record Part IV*. Institute of Radio Engineers, New York, pp. 96–104 (cit. on p. 25).
- Wiering, M. and J. Schmidhuber (1998). “Efficient Model-Based Exploration”. In: *Proceedings of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats 5*, pp. 223–228 (cit. on pp. 29, 63, 64).
- Wiering, M. and H. van Hasselt (2008). “Ensemble algorithms in reinforcement learning.” In: *IEEE transactions on systems, man, and cybernetics* 38.4 (cit. on p. 80).
- Zimmermann, H.-G., R. Grothmann, A. Schäfer, and C. Tietz (2006). “Modeling Large Dynamical Systems with Dynamical Consistent Neural Networks”. In: *New Directions in Statistical Signal Processing: From Systems to Brain*. Ed. by S. Haykin, J. Principe, T. Sejnowski, and J. McWhirter. MIT Press, pp. 203–242 (cit. on p. 80).



# Index

- $\xi$ -optimal policy, 38
- action-value function, 16
- agent, 13
- agent self-assessment, *see* policy assessment
- archery benchmark, 51
- bagging, 79
- Bayesian estimator, 34
- Bayesian Q-learning, 63
- Bellman equation, 17
- Boltzmann action selection, 29
- boosting, 79
- border phenomenon, 45
- cart-pole benchmark, 82, 118
- changing environment, 109
- chattering in NFQ, 73
- continuous-action problems, 91
- covariance matrix, 40
- data-efficiency, 23, 25
- diagonal approximation of uncertainty-incorporating policy iteration, *see* DUIPI
- Dirichlet distribution, 34, 48
- Dirichlet prior, 47
- discount factor, 13
- DUIPI, 41, 43, 49
  - with Q-modification, *see* DUIPI-QM
- DUIPI-QM, 49
- dynamic programming, 18
- eligibility traces, 24
- ensembles, 78
  - accuracy and ambiguity, 78
  - evolving, *see* evolving ensemble
  - weighted, 103
- environment, 13
- estimator
  - reward, 32
  - transition probabilities, 32
- evolutionary algorithm, 28
- evolving ensemble, 109, 112
- expected return, 55
- experience replay, 23
- exploration, 28, 63
  - $\epsilon$ -greedy, 29
  - (un-)directed, 29
  - random, 28
  - uncertainty-seeking, 45
- extremely randomized trees, 100
- fitted policy evaluation, 96
- fitted policy search, 62
- fitted Q-iteration, 26
  - neural, *see* NFQ
- fitted value iteration, 26
- frequentist estimator, 33
- full-matrix uncertainty propagation, 39, 49
- function approximation, 25, 71
- Gaussian error propagation, *see* uncertainty propagation
- gradient descent, 25
- greedy region, 73
- grid discretization, 71
- imbalanced data problem, 113
- inverted pendulum, *see* pole-balancing benchmark
- majority voting, 78
  - weighted, 103
- Markov decision process, 14
  - partially observable, 15
- Markov property, 14
- maximum entropy prior, 34
- MDP distribution, 32
- model-based interval estimation, 29, 63
- Monte Carlo sampling, 96
- Monte Carlo uncertainty estimation, 36
- most agreeable policy, 81, 106

- most preferable policy, 106
- multinomial distribution, 33, 34, 48
- neural fitted Q-iteration
  - see NFQ, 71
- neural network training, general issues, 76
- NFPE, 98
- NFQ, 71
  - problems of, 72
- optimal policy, 17
- overestimation of  $Q$ -values, 74
- pole-balancing benchmark, 83, 110, 113
- policy, 13
- policy combination in ensembles, 80
- policy degradation in NFQ, 61
- policy evaluation, 19, 48
- policy gradient, 27
- policy improvement, 19
- policy iteration, 19
- policy pool, 110, 112
- policy rejection, 101
- policy search, 28
- policy selection, 101
  - in changing environments, 110
- POMDP, *see* Markov decision process, partially observable
- prioritized sweeping, 24
- Q-function, 16
- Q-learning, 23
- Q-value distribution, 32
- quality assurance, 31, 45
- quantile optimality, 31
- quantile performance, 32
  - maximization, 38
- quantile-optimal policy, 31
- R-Max, 63
- relative frequency, 33
- resampling of observations, 113
- return, 13
- rewards regression, 26
  - kernel, 26
  - neural, 26
- rising Q problem, *see* overestimation of  $Q$ -values
- risk in RL, 65
- river-swim benchmark, 65
- SARSA, 22
- self-assessment, 54
  - in continuous domains, 95
  - value function-based, 55, 95
- sequential decision making, 13
- sigmoid activation function, 74
- state aggregation, 25
- state estimator, 15
- state-value function, 16
- temporal-difference methods, 22
- trap benchmark, 56, 65
- TreeFPE, 100
- uncertainty awareness, 31
  - applications, 45
- uncertainty propagation, 31, 39
- uncertainty-aware value iteration, 38
- value function, 15, 16
  - as indicator of policy quality, 55, 98
  - quantile, 56
- value iteration, 20, 47
- VarioEta, 75
- weighted ensemble, 110
- wet-chicken benchmark, 51, 84